```
***********************************************************
*                                                         *
*                                                         *
*                                                         *
*          USL / DBMS    NASA / PC    R&D                 *
*                                                         *
*                                                         *
*          WORKING    PAPER    SERIES                     *
*                                                         *
*                                                         *
*                Report Number                            *
*                                                         *
*                                                         *
*            DBMS.NASA/PC R&D-20                          *
*                                                         *
*                                                         *
*                                                         *
*                                                         *
***********************************************************
```

For more information, contact:

Wayne D. Dominick

Editor
USL/DBMS NASA/PC R&D Working Paper Series
Center for Advanced Computer Studies
University of Southwestern Louisiana
P. O. Box 44330
Lafayette, Louisiana 70504
(318) 231-6308

(NASA-CR-184552) A PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY FOR OBJECT-ORIENTED SYSTEMS Ph.D Thesis Final Report, 1 Jul. 1985 - 31 Dec. (University of Southwestern Louisiana. Lafayette. Center for Advanced   N89-14992   Unclas   G3/82   0183590

Consequently, the research presented in this document addresses the design, development, and evaluation of a systematic, extensible, and environment-independent methodology for the comparative evaluation of object-oriented programming environments. This methodology is intended to serve as a foundational element for supporting research into the impact of object-oriented software development environments and design strategies on the software development process and resultant software products. A systematic approach is defined for conducting the methodology with respect to the particular object-oriented programming environment under investigation. The evaluation of each environment is based on user performance of representative and well-specified development tasks on well-characterized applications within the environment. Primary metrics needed to characterize the software applications under examination are also defined and monitored for subsequent use in the analysis and evaluation of the environments.

The major contributions of this work are as follows:

1. This research has formally established the primary metric data definitions that completely characterize the unique aspects of object-oriented software systems, including the inheritance lattice and messaging graph.

2. This research has established language-independent procedures for automatically capturing this primary metric data during an evaluation. These procedures have been shown to be instantiable in a representative set of object-oriented languages.

3. This research has established the fundamental characteristics of object-oriented software that indicate consistent applications of object-oriented design techniques, namely, that common capabilities are factored throughout the inheritance lattice and that individual objects focus on providing specific capabilities.

4. This research has defined a language-independent application domain-specific development paradigm based on these fundamental characteristics for highly interactive graphical applications.

5. This research has identified design principles for a programming environment evaluation methodology (PEEM) that ensure its applicability to object-oriented development environments. The PEEM design principles unique to this work include the following: the requirement for primary metric data definitions that completely characterize the object-oriented characteristics of the software under evaluation, the requirement for the identification of relevant applications domain-specific development paradigms to support the validity and comparability of evaluative results, and the requirement for automatic capture of performance and primary metric data to ensure consistency and eliminate human bias.

6. Finally, this research has produced a systematic, extensible, and environment-independent programming environment evaluation methodology capable of supporting research into complexity models and metrics for object-oriented systems. The design principles, identified in contribution˜5 above, establish the basis of the fundamental distinctions between exiting PEEMs and the PEEM developed as part of this research.

This report represents one of the 72 attachment reports to the University of Southwestern Louisiana's Final Report on NASA Grant NGT-19-010-900. Accordingly, appropriate care should be taken in using this report out of the context of the full Final Report.

A PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY FOR OBJECT-ORIENTED SYSTEMS

A Dissertation

Presented to

The Graduate Faculty of

The University of Southwestern Louisiana

In Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Dennis R. Moreau

September, 1987

A PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY FOR OBJECT-ORIENTED SYSTEMS

Dennis R. Moreau

APPROVED:

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to Professor Wayne D. Dominick for chairing my committee and for supporting and guiding this research. His encouragement, commitment, professionalism, responsiveness and friendship have made this work possible. I would also like to thank Dr. William R. Edwards and Dr. Steve P. Landry for providing their valuable assistance and serving on my committee.

Several other researchers with whom I have had insightful discussions concerning the significance and progress of this research should also be acknowledged. They include John Eubbing and Bob Walstra of Hewlett-Packard's Palo Alto Research Laboratories, Kurt Shmucker of Productivity Products International and Zack Shorer of AT&T Bell Laboratories.

The author would like to acknowledge the significant support provided to him throughout the duration of these research activities from the National Aeronautics and Space Administration via a number of grants and contracts, including NASA Contract Number NASW-3846, NASA Research Grant Number NAGW-701, and NASA Training Grant Number NGT-19-010-900.

The author would also like to acknowledge certain key individuals whose assistance in developmental activities through participation in the USL NASA Project proved invaluable. They are Brijesh Agarwal, Peter C. Bahrs, John M. Love, Desiree J. Matta, Patrick R. Michaud, and Kam C. Wong.

Finally, the author would like to acknowledge AT&T Information Systems, Inc. for donating to the USL NASA Project the hardware environment that has served as the set of controlled-usage workstations for supporting the specific test case implementation and experimentation aspects of this research, and to acknowledge the 110+ different organizations whose donated software to the USL NASA project has provided an exceptional software development environment for these activities:

1. International Business Machines Corporation
2. Lifeboat Associates
3. SPSS, Incorporated
4. Phoenix Computer Products Corporation
5. Applied Technical Systems
6. Wiley Professional Software
7. Rational Systems, Inc.
8. Arity Corporation
9. Central Point Software Incorporated
10. Spruce Technology Corporation
11. Gold Hill Computers
12. UniPress Software
13. The Software Link, Incorporated
14. Oracle Corporation

15. Relational Database Systems, Inc.
16. Computer Innovations, Inc.
17. Greenleaf Software, Inc.
18. Emerging Technology Consultants, Inc.
19. Microrim, Inc.
20. WELCOM Software Technology
21. Spartacus, Inc.
22. FTG Data Systems
23. Strategic Software Planning Corporation
24. Penton Software, Inc.
25. Zanthe Information, Inc.
26. Expertware, Inc.
27. Productivity Products International, Inc.
28. Production Systems Technologies, Inc.
29. Network Software Associates, Inc.
30. The Lisp Company
31. IMSL Incorporated
32. MacMillan Software Company
33. Simulation Software, Ltd.
34. Alloy Computer Products, Inc.
35. Texas Instruments, Incorporated
36. Microsoft Corporation
37. Lotus Development Corporation
38. FTP Software, Inc.
39. Foresight Resources Corporation
40. Micrografx, Inc.
41. Gimpel Software
42. Lugaru Software, Ltd.
43. Enertronics Research, Inc.
44. Prospero Software
45. American Small Business Computers
46. McDermott Corporation
47. WordPerfect Corporation
48. Software Architecture & Engineering, Inc.
49. AT&T Information Systems, Inc.
50. Locus Computing Corporation
51. UNIFY Corporation
52. Touchstone Software Corporation
53. Rhodnius Incorporated
54. The Pilot Group, Intelligent CourseWare, Inc.
55. BRS Information Technologies
56. Conetic Systems, Inc.
57. Aker Corporation
58. Prior Data Sciences Product Sales, Inc.
59. Systems Compatibility Corporation
60. HavenTree Software Limited
61. Paul Mace Software, Inc.

62. InstaPlan Corporation
63. The Whitewater Group
64. Datalight
65. MathSoft, Inc.
66. Personal TEX, Inc.
67. Kemp-Carraway Heart Institute
68. MicroPro International Corporation
69. POWER-BASE Systems, Inc.
70. mbp Software and Systems Technology, Inc.
71. Gazelle Systems
72. Software Publishing Corporation
73. MicroControl Systems, Inc.
74. Scientific Endeavors Corporation
75. Persoft, Inc.
76. Expert Systems International
77 AT&T Bell Laboratories (Murray Hill Labs)
78. AT&T Bell Laboratories (Morristown Labs)
79. Datastorm Technologies, Inc.
80. Catspaw, Inc.
81. Micro Data Base Systems, Inc.
82. Ron Turley Computer Associates
83. Peachtree Software
84. EXSYS, Inc.
85. Universal Technical Systems, Inc.
86. CalComp
87. Image Network
88. The Santa Cruz Operation
89. Symantec Corporation
90. R.R. Software, Inc.
91. Custom Software Systems
92. DataEase International
93. True BASIC, Inc.
94. JMI Software Consultants, Inc.
95. Automata Design Associates
96. The Wollongong Group
97. MIX Software
98. The Small Computer Company, Inc.
99. Generic Software, Inc.
100. UniPress Software
101. Data Language Corporation
102. Integral Quality, Inc.
103. Communications Research Group
104. STSC, Inc.
105. Mouse Systems Corporation
106. MicroProducts, Inc.
107. Access Technology, Inc.
108. Ashton-Tate

109. General Research Corporation
110. Campbell Services, Inc.
111. Meridian Software Systems, Inc.
112. Interactive Systems, Inc.
113. Nastec Corporation

The hardware and software environments, donated by these organizations and integrated into the NASA PC R&D Laboratory under the auspices of the USL NASA Project, have provided an extremely robust and comprehensive developmental and experimental laboratory for supporting the object-oriented systems aspects, the automatic monitoring and program metrics aspects, and the graphics aspects of this research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: THE PROBLEM

The object-oriented design strategy as both a problem decomposition and system development paradigm has made impressive inroads into the various areas of the computing sciences. Substantial development productivity improvements have been demonstrated in areas ranging from artificial intelligence to user interface design. But, formally characterizing these productivity improvements and identifying the underlying cognitive mechanisms remain as research tasks. There is no formal cognitive approach that adequately models traditional software development, let alone one that models software development under non-traditional paradigms [JonesC 1986, Harrison 1985]. The task at hand, then, is to attempt to develop such a model and the associated complexity metrics for object-oriented systems. The development and validation of models and metrics of this sort require large amounts of systematically gathered structural and productivity data [Harrison 1985]. There has, however, been a notable lack of systematically gathered information on these development environments. A large part of this problem is attributable to the lack of a systematic programming environment evaluation methodology that is appropriate to the evaluation of object-oriented systems.

Such a methodology is critical to the reliable collection of performance data and software characteristics, and to the effective dissemination of this information in support of long-term research into software complexity metrics for object-oriented systems. It is this long-term complexity research that holds the promise of identifying just why object-oriented development environments are so effective in increasing software productivity in a wide variety of applications areas.

Consequently, the research presented in this document addresses the design, development, and evaluation of a systematic, extensible, and environment-independent methodology for the comparative evaluation of object-oriented programming environments. This methodology is intended to serve as a foundational element for supporting research into the impact of object-oriented software development environments and design strategies on the software development process and resultant software products. A systematic approach is defined for conducting the methodology with respect to the particular object-oriented programming environment under investigation. The evaluation of each environment is based on user performance of representative and well-specified development tasks on well-characterized applications within the environment. Primary metric data needed to characterize the software applications under test is also collected for subsequent use in the analysis and evaluation of the environments. These metrics are not intended as complexity measures themselves, but rather as systematic indexes of software characteristics.

1.1 Object-Oriented Systems Defined

Despite the recent widespread emergence of object-oriented systems technology in widely diverse software domains, there is a strong consensus among researchers as to the characteristics of a truly object-oriented system [Agha 1986, Cox 1986, Meyer 1987, Kaehler 1985, Schmucker 1985, Wegner 1986]. According to this consensus, object-oriented systems are those which include an inheritance mechanism for module construction, data and procedure encapsulation, typed messaging for module invocation, and some form of late-binding of messages to target modules. A consensus has also formed with regard to object-oriented systems terminology. This section establishes these fundamental definitions as a prelude to a survey of object-oriented systems activity in several areas.

An object consists of private data and a set of operations that can access that data. An object is requested to perform an operation by sending it a typed message. A particular method (operation) is invoked based on the type of the incoming message. In pure object-oriented systems, this is the only way to invoke an object's method and is consequently the only way to change an object's state. The messaging mechanism is responsible for associating typed messages with the appropriate objects. A class is an abstract object type. All objects of a class have data and method characteristics in common. The inheritance mechanism provides a means of constructing new object classes from existing classes. Only the differences between the existing class(es) and the new class need be specified.

There are two primary varieties of inheritance mechanisms, the single inheritance mechanism, which allows only one parent class per object, and the multiple inheritance mechanism in which this restriction is not observed. This characterization of object-oriented systems and the associated definitions have emerged as a result of the efforts of many researchers and are not attributable to any one person. However, many of the ideas and much of the foundational development in this area is attributable to Adele Goldberg (SMALLTALK) [Goldberg 1983] and Kristen Nygaard (SIMULA) [Dahl 1966].

1.2 Overview of Existing Object-Oriented Systems Efforts

This section overviews the development of object-oriented systems technology in widely diverse areas of the computing sciences. This overview is intended to provide a motivation for, and preliminary indication of the potential for research in this area. It is the opinion of many researchers in software engineering that object-oriented design strategies and development environments are the most promising approaches to increasing productivity on the horizon [JonesC 1986, Meyer 1987, Cox 1986].

1.2.1 The User Interface Perspective

It is likely that no other applications area is more closely associated with object-oriented systems technology than that of user interface design and implementation. Ironically, the two topics are only peripherally connected. There is certainly no requirement that object-oriented systems have good user interfaces, and one can certainly build traditional user interfaces in an object-oriented development environment such as SMALLTALK. Also, it is conceivable (just barely) that a windowing iconic user interface could be constructed in a language like COBOL. The nature of the connection between these two issues lies in the ability of object-oriented systems to provide complexity management mechanisms to the user interface developer including facilities for module re-use and encapsulation.

Work conducted by numerous researchers including Brad Cox of Productivity Products International [Cox 1986], Norman Meyrowitz of Brown University [Meyrowitz 1986], David Anderson of Carnegie-Mellon University [Anderson 1986], and Daniel Bobrow of the Xerox Palo Alto Research Center [Bobrow 1986] has demonstrated productivity improvements of 300 to 600 percent over traditional approaches to developing complex user interfaces. It is no wonder that graphical, iconic, windowing user interfaces came to be known as object-oriented user interfaces. This sort of user interface, together with a generous portion of object-oriented systems technology, has been popularized in the current crop of consumer-oriented personal computers, including the Apple Macintosh, Atari ST, Commodore Amiga, and IBM PCs running Microsoft Windows.

1.2.2 The Simulation Perspective

Simulation is probably the applications area most legitimately associated with object-oriented systems technology. It can be argued that object-oriented design had its start in SIMULA, a language specifically designed to support simulation development. While simulation applications benefit strongly from the same complexity management facilities that support user interfaces so well, simulation systems were able to capitalize on an additional benefit characteristic to object-oriented systems, namely, simulation systems were able to make use of a reduction in the semantic gap between the system to be simulated and the simulation software. This is due to the fact that objects (modules) in object-oriented systems communicate by sending messages to other objects. This provides a very flexible way of structuring simulation software; so flexible in fact, that it very closely resembles the real system being simulated.

Research by Birtwistle [Birtwistle 1984], Franta [Franta 1973], Papazoglou [Papazoglou 1984], and Kreutzer [Kreutzer 1986] has established object-oriented design as the technique of choice for discrete event simulation and for combined discrete-continuous simulation approaches. Object-oriented simulation languages in these categories include SIMULA [Dahl 1966], SIMON [Sim 1975], DEMOS [Kreutzer 1986], and DISCO [Helsgaun 1980]. However, other languages including

SMALLTALK [Goldberg 1983], GLISP [Novak 1983], and Flavours [Weinreb 1981] are being used extensively primarily due to their support for multiple inheritance and animated graphical presentation.

1.2.3 The DBMS Perspective

Database management systems researchers, in addition to capitalizing on the complexity management features and natural modeling characteristics of object-oriented software, have been able to make use of the messaging mechanism to model the usage of distributed resources. This has led to a flurry of DBMS activity in the CAD/CAM area. Notable representative research includes that of Maier of Servio Logic Development Corporation [Maier 1986], and Skarra of Brown University [Skarra 1986]. CAD/CAM applications are highly dynamic by virtue of the rapidly changing technologies employed and so make good use of the semantic similarity between object-oriented database designs and the "real" application. These systems are highly flexible and support runtime re-configuration of the database while providing high query efficiency due to the practical locality of item references. That is, items that are closely related in the real system are closely related in the database design so natural access paths are explicitly captured in the object-oriented DBMS implementation.

1.2.4 The Artificial Intelligence Perspective

Object-oriented development facilities have been commonplace in artificial intelligence development environments for some time. These facilities provide integrated and consistent access to AI-workstation resources and support highly productive software development activities [Moon 1986], but the most significant use of object-oriented techniques within AI environments lies in the area of knowledge base organization.

In these systems, knowledge is typically organized as a collection of implementation-independent communicating objects. This organization facilitates modularization of the knowledge base. The inheritance mechanism enables the incremental development of new objects by specialization of existing object types, supporting a factorization of knowledge as a class hierarchy [Stefik 1986].

These capabilities have led to the inclusion of object-oriented features in a large number of AI development languages. There are widely used object-oriented logic programming languages including SPOOL [Fukunaga 1986] and Intermission [Fukunaga 1986]. Object-oriented extensions to LISP include LOOPS [Bobrow 1986], FLAVORS [Moon 1986], and SCHEME [Lang 1986]. SMALLTALK [Goldberg 1983] and Orient84/K [Ishikawa 1986] provide a basis for AI development in distributed computing environments, as does Carl Hewitt's ACTOR [Hewitt 1973]. Finally, many knowledge engineering and expert system development environments include object-oriented capabilities as primary features. These include ESP [Chikayama 1984], KEE [Fikes 1985] and URANUS [Nakashima 1984].

1.2.5 The Operating Systems Perspective

Emphasizing the encapsulation, polymorphism, and dynamic binding aspects of object-oriented systems, several operating systems have been developed using object-oriented techniques, including MACH [JonesM 1986], CLOUDS [Dasgupta 1986] and Emerald [Black 1986]. The dynamic binding aspect of object-oriented messaging supports dynamic reconfiguration of distributed resources and can be supported on top of existing network services facilities. The encapsulation aspects support the integration of multiple heterogeneous distributed environments in a consistent and extensible fashion.

In MACH, this is embodied in the port, message and memory kernel abstractions that are consistently modeled on all member distributed systems. Generic messages that control distributed tasks are mapped into the appropriate system service requests on the receiving system. The computational objects, tasks and their constituent threads can be distributed to distinct servers for execution, since they also issue and respond to generic object messages. Even presentation facilities are modeled as object-oriented resources in that user interface windows, graphical and textual capabilities can be inherited and distributed across available resources.

This research has much in common with the DBMS research referred to in Section 1.2.3 and holds equal promise for providing highly functional and flexible distributed heterogeneous computing environments.

1.2.6 The General Applications Development Perspective

Recent object-oriented systems research has focused on migrating the capabilities cited in the sections above into the general software development domain. These efforts have demonstrated significant progress as indicated by the degree of commercial interest and involvement in using object-oriented software development systems for both internally-used and commercially-marketed products. The languages that fall into this category include Productivity Products International's Objective-C [Cox 1986], AT&T's C++ [Stroustrup 1986], and Interactive Software Engineering's EIFFEL [Meyer 1987].

These languages are hybrid in that they provide linkages to traditional languages and traditionally-developed software libraries. This preserves investments in, and capitalizes on, the large base of existing software facilities. These languages bring object-oriented techniques to the traditional UNIX software development environment and have made possible productivity gains of about 5-to-1 for a large variety of application domains [Cox 1986, Stroustrup 1986].

Although this work has been generally confined to C and UNIX environments, the increases in productivity that have been experienced have raised interest in object-oriented extensions to other general purpose languages.

The areas overviewed above are representative of the scope of the work currently being conducted in object-oriented systems and do indicate clear and substantial evidence for the advantages of this approach. It is this evidence that motivates the research presented in this document.

1.3 General Research Objectives

The following are the general research objectives identified for this research:

1. The design, development, application, and evaluation of a systematic, extensible, and environment-independent evaluation methodology capable of supporting investigation into the impact of object-oriented design strategies on the software development process.

Existing approaches to programming environment evaluation have serious shortcomings in the context of evaluating object-oriented systems. Approaches developed for traditional software development environments have no provision for capturing and recording software characteristics unique to object-oriented software [Weiderman 1987]. Approaches based on criteria checklists (typical of the Ada environment evaluation methodologies) [Brinker 1985, Castor 1983, Hook 1985] do not provide adequate detail and rigor to support research into software complexity metrics. None of the existing traditional approaches provide for inter-paradigm comparison and evaluation. The primary justification of this general objective, then, is the lack of an existing methodology appropriate to the evaluation of object-oriented systems and to the support of complexity metrics research. A representative application of this methodology forms the basis of an analytical evaluation and demonstration of its capabilities. Comprehensive validation of this methodology will require long-term usage, beyond the scope of this research.

2. The design, development, application, and verification of domain-specific applications development paradigms to support consistent comparisons of applications developed under an object-oriented strategy.

Domain-specific application development paradigms are sets of problem decomposition and solution guidelines that are appropriate to certain applications domains under certain software technologies. For example, the development of 2-D vector graphics applications typically incorporates a decomposition at the highest level into four components, namely, the user interface, the device interface, the file system interface, and the application interface. A specific instance of this paradigm for a graphical kernel system (GKS) graphics library would include the workstation interface, the virtual device interface (VDI), the metafile system, and the specific language binding. These application development paradigms guide software into appropriate organizations. Other common application development paradigms include recursive descent organization for parsers, protocol layering for communications and networking software, and various organizational strategies for user interface development.

It is clearly possible to use an inappropriate application organization strategy to develop object-oriented applications. This would result in applications that do not benefit from object-oriented design techniques and so would not exhibit the productivity characteristics of object-oriented systems, obscuring any meaningful comparison between the "object-oriented" and non-object-oriented applications. The intent of this general objective is to identify specific design characteristics that characterize object-oriented applications and to design development paradigms that promote those characteristics for specific applications domains. The evaluation of these paradigms consists of an analysis of how the overall design space for a specific application is constrained by the paradigm.

Existing programming environment evaluation methodologies do not even make provisions for recording the specific development paradigm used. The presented methodology incorporates the application-specific paradigm as a means of providing consistency within applications under consideration.

3. The design, development, application, and completeness verification of primary metric data definitions appropriate to systems developed under an object-oriented design strategy.

In order to perform comparative evaluations of competing object-oriented systems, one must have information on the differences between the characteristics of the code produced using the respective systems. This information is also necessary for supporting software complexity metric research. For the purposes of this research, primary metric data is defined as the minimal set of particular software characteristics needed to fully characterize the unique aspects of object-oriented systems, together with those measurements needed to support comparisons using accepted traditional metrics (e.g., McCabe's cyclomatic complexity [McCabe 1976], Halstead's software science metrics [Halstead 1977], etc.). While there has been some work performed on providing comprehensive primary metric data definitions for traditionally-developed software, this work is still exploratory [Harrison 1985]. No existing programming environment evaluation methodologies provide for the characterization of object-oriented systems.

The procedures developed for capturing this data for specific object-oriented environments is comprehensively evaluated for accuracy and coverage across existing object-oriented language features.

CHAPTER 2: SPECIFIC RESEARCH OBJECTIVES

In this chapter, the general research objectives of Section 1.3 are refined into constituent specific research objectives. The primary intent of this research is to provide a framework for the investigation of the impact of object-oriented software development environments and design strategies on the software development process and resultant software products.

2.1 Specific Refinements of the Evaluation Methodology General Research Objective

Evaluation Methodology General Research Objective: The design, development, application, and evaluation of a systematic, extensible, and environment-independent evaluation methodology capable of supporting investigation into the impact of object-oriented design strategies on the software development process.

The specific research objectives identified pursuant to the satisfaction of the evaluation methodology related general research objective include the following:

A. *Specific theoretical objective:* to develop design principles for the proposed evaluation methodology that ensure systematic, reproducible, and environment-independent performance evaluations, thereby supporting the long-term development of theoretical models and metrics for the characterization and comparison of object-oriented systems.

These design principles provide the theoretical foundation for the proposed methodology. The evaluation of the methodology is based in part on the degree to which it adheres to these principles.

B. *Specific methodological objective:* to design an evaluation methodology that incorporates automatic performance and primary metric data collection. The practicality of any evaluation methodology is determined by the relevance of its results and, to a lesser extent, by the cost of its execution.

The automatic data collection characteristic of the methodology eliminates the high cost and inherent unreliability of manual data collection within the evaluation process, while focusing emphasis on user performance data as an evaluation criteria provides results that are directly relevant to the needs of the organization conducting the evaluation.

C. *Specific developmental objective:* to develop a prototype evaluation environment capable of supporting the proposed methodology.

This environment is evaluated based on its support for the specific requirements of the proposed evaluation methodology, including unobtrusive automatic performance data collection and primary metric computation and recording.

D. *Specific evaluative objective:* to conduct a systematic comparison of selected software development tasks in a specific applications domain using an object-oriented programming system with the same development tasks using a traditional programming system under the proposed methodology to demonstrate its evaluation strategy and the capabilities of this approach.

The intent of this evaluation process is to provide a demonstration of the ability of this methodology to support evaluations across heterogeneous programming environments. This intent is reflected in the relatively small number of subjects selected for participation in the evaluation process (see Section 4.7).

Significance: Existing system evaluation methodologies make no provision for cross-environment evaluation. This prohibits use of these methodologies in comparing object-oriented versus non-object-oriented systems. The research literature in object-oriented systems is notably devoid of systematically collected data and structured analytical results. This methodology addresses these problems specifically.

2.2 Specific Refinements of the Application-Specific Paradigm General Research Objective

Application-Specific Paradigm General Research Objective: The design, development, application, and verification of domain-specific applications development paradigms to support consistent comparisons of applications developed under an object-oriented strategy.

The specific research objectives identified pursuant to the satisfaction of the application-specific development paradigm related general research objective include the following:

A. *Specific theoretical objective:* To determine fundamental design characteristics for specific applications domains that promote the theoretical validity of systematic comparisons of object-oriented systems.

If these characteristics are present in the applications under consideration, confidence in the generalizability of the performance comparisons is enhanced, since the applications are truly representative object-oriented designs.

B. *Specific methodological objective:* To design applications domain-specific paradigms that support the effective application of object-oriented design techniques.

The applications domain-specific development paradigms are intended to guide software development so that the resultant product exhibits the organizational characteristics referred to in Specific Objective A above. The evaluation of this specific objective is based on the degree to which this paradigm ensures these characteristics.

C. *Specific developmental objective:* To develop procedures for the application of the applications domain-specific paradigms within a specific object-oriented development environment.

These procedures are the instantiation of the paradigms referred to in Specific Objective B above, for a specific object-oriented programming environment. These procedures are evaluated for their accuracy in representing the application-specific paradigms and for the degree to which they make use of, or preclude features within the specific environment.

D. *Specific evaluative objective:* To analytically verify that the applications-specific development paradigm does indeed constrain the resultant software products so that they are indeed representative of object-oriented designs.

This evaluation is based on an analysis of the software generated under the application-specific paradigm and the degree to which it exhibits the fundamental characteristics of an object-oriented design.

Significance: It is certainly possible to develop applications using traditional design strategies within an object-oriented environment. This produces applications that are not representative of object-oriented designs and invalidates any meaningful comparison for our purposes. The application domain-specific paradigms proposed under this research objective provide problem decomposition and application development guidelines that, when applied, will ensure that the application is indeed representative of an object-oriented design. These types of guidelines do exist for traditionally-developed software in specific domains and include standard system organizations for graphics systems software, compiler construction, network system software, operating system software, and various user interface organizations.

2.3 Specific Refinements of the Primary Metrics General Research Objective

Primary Metrics General Research Objective: The design, development, application, and completeness verification of primary metric data definitions appropriate to systems developed under an object-oriented design strategy.

The specific research objectives identified pursuant to the satisfaction of the primary software metrics related general research objective include the following:

A. *Specific theoretical objective:* To design primary metric data definitions that theoretically characterize the various aspects of software unique to object-oriented designs, including the inheritance lattice, the messaging graph, the degree of polymorphism exhibited, and degree of object re-use.

This primary metric data is necessary to describe the structural aspects of software unique to object-oriented systems. The evaluation of this objective consists of a completeness verification of these defintions, that is how completely the primary definitions capture these unique characteristics.

B. *Specific methodological objective:* To develop language-independent methods for capturing this data for object-oriented designs. Language independence is demonstrated by constructing language-specific metric evaluation procedures for a representative set of object-oriented languages.

Consistent capture of the primary metric data requires language-independent specification of the respective acquisition procedures. The evaluation of this objective is based on the ability of these language-independent methods to be instantiated in a representative set of existing object-oriented languages as language-specific procedures.

C. *Specific developmental objective:* To provide language-specific acquisition of this metric data for the object-oriented development systems under consideration in the evaluation environment.

This objective provides the language-specific mechanism for supporting the automatic primary metric data collection aspects of the evaluation methodology general objective.

D. *Specific evaluative objective:* To comprehensively test these metric data acquisition methods for accuracy.

This objective provides confidence in the automatic primary metric data acquisition mechanisms. The accuracy of these mechanisms is essential to conducting valid complexity metric research.

Significance: Existing software metrics are not appropriate to comprehensively characterizing object-oriented systems for several reasons. The most accepted (and validated) metrics are motivated by assumptions about the relationship between program size and programmer productivity that are not directly valid in object-oriented systems due to code re-use. Traditional metrics do not address the structural aspects of software characteristic of object-oriented systems, including the organization of the inheritance lattice, the organization of the messaging graph, and module re-use characteristics. In traditional metrics research, the complexity of very large systems is often viewed as an extrapolation of the complexity of its constituent components; however, inheritance and encapsulation in object-oriented systems have the demonstrated effect of reducing individual module size and external coupling to the extent that very large system complexity is not clearly an extension of the aggregate complexity of the constituent objects [Cox 1986, Stroustrup 1986, Meyer 1987].

As part of the final evaluation of this research, Table 5.1 indicates the mapping between the specific research objectives identified in this chapter and the specific sections in Chapter 5 which evaluate the degree to which each respective objective is accomplished.

CHAPTER 3.0: THE PROPOSED PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY

This chapter overviews representative programming environment evaluation methodologies and analyses these methodologies with respect to their ability to support the research objectives of Section 1.3.

3.1 Existing Programming Environment Evaluation Methodologies

Very little work has been done in the area of programming environment evaluation methodologies. The majority of work in this area has focused, not surprisingly, on Ada development environments. The work that does exist in this area has taken three major approaches.

The first approach is based on extensions of evaluation work performed in traditional applications areas including DBMS evaluation, user interface evaluation, compiler evaluation, and, most prolifically, text editor evaluation. Notable research of this type in the area of programming environment evaluation includes Lindquist's "Assessing the Usability of Human-Computer Interfaces" [Lindquist 1985]. However, this evaluation approach focuses on the individual tools within an environment without considering how well the tools are integrated.

A second approach in the literature, exemplified by Brinker's "An Evaluation of the Softech Ada Language System" [Brinker 1985], is to focus on specific program development environments. While work of this type does make explicit provision for evaluating the overall environment, no provision is made for inter-environment comparisons. This work tends to focus on system-specific criteria and metrics that are very difficult to evaluate consistently across heterogeneous environments.

A third approach includes research that proposes lists of often subjective criteria, concerning features and facilities of the environments in question. While there is a characteristic effort to construct very comprehensive sets of criteria, these criteria still tend to be rather system and domain specific. There have been some surprising results from methodologies incorporating this sort of approach, including the COCOMO work [Conte 1985] and Productivity Research Incorporated's Software Productivity, Quality, and Reliability (SPQR) model [JonesC 1986]. The SPQR technique has demonstrated a predictive accuracy of within 15% for software productivity and quality over a large number of tests for very large systems development efforts [JonesC 1986]. Unfortunately, these techniques are often difficult to apply consistently and are all but impossible to automate. There is work that suggests that each criteria be given an operational definition to permit automatic evaluation [Bailey 1985]; however, this approach has come under substantial criticism, since the operational definitions of subjective criteria are themselves subjective interpretations of how these criteria should be measured [Weiderman 1987].

Recent work at CMU's Software Engineering Institute has resulted in a hybrid approach, combining traditional techniques with evaluative questionnaires constructed by "experienced" evaluators. This technique makes provision for experimentation as well as subjective evaluation, but makes no provision for ensuring the representativeness of the test applications with regard to their respective design strategies. This technique would be difficult to apply across a large and diverse evaluation base due to its reliance on "experience" in questionnaire formulation and this technique may suffer from consistency problems in application due to the subjective nature of many of its criteria. Validation of this approach is currently under way [Weiderman 1987].

3.2 Problems with Existing Methodologies

Each of the programming environment evaluation strategies referred to in Section 3.1 is inadequate for supporting the general research objectives cited in Section 1.3. Each of these approaches falls short in at least one of the following areas:

1. Certain of the existing approaches focus on particular tools and, in doing so, ignore the overall impact of the environment on software development. While the information gathered in these approaches is valuable in characterizing and evaluating individual tools, the information so gathered is of little use in evaluating the much more complex programming environments where the interaction between support facilities is a major factor in development productivity. For example, while C++ and C in a UNIX environment each provide tools of similar functionality, the degree of integration and coordination of these facilities accounts for a fivefold productivity improvement of C++ over C [Cox 1986, Stroustrup 1986, Meyer 1986, JonesC 1986].

2. Certain of the existing approaches make assumptions that preclude or make no provision for evaluations of systems across highly heterogeneous environments. This is primarily due to the lack of correspondence between tools and facilities in the competing environments. There is

simply no comparison between the facilities provided by SMALLTALK [Goldberg, 1983] and the facilities typically provided to FORTRAN programmers. The only common ground in this situation (in the absence of a generically-valid cognitive complexity model) is user performance within the respective environments for tasks accomplishable in both.

3. Certain of the existing approaches rely on sets of expertly-defined evaluation criteria as a basis for programming environment evaluation. These criteria tend to be subjective in nature and are characteristically environment-specific. They do not lend themselves to automation and so retain human bias and inconsistency. It is difficult to imagine a set of evaluation criteria that would provide an objective evaluation of Objective-C [Cox 1986] versus SMALLTALK, since any set of criteria would be either incomplete for SMALLTALK or primarily irrelevant for Objective-C. Even if such a set of criteria could be established, the results of a comparative evaluation of these two environments using this approach would consist of a criteria check list that would not support the development of formal models for the complexity of object-oriented systems.

4. All of the existing approaches fail to identify minimal primary metrics relevant to characterizing the structure of object-oriented software systems. This aspect of the evaluation methodology is critical to supporting long-term research into software complexity metrics for object-oriented systems.

In summary, the existing approaches to programming environment evaluation are not appropriate to supporting the evaluation of object-oriented systems. The research presented within this document focuses on this issue.

3.3 Design Principles of the Proposed Methodology

The evaluation of software development environments is a difficult problem primarily because of the wide variation in the available sets of tools in respective environments and the levels of design abstraction at which these tools are aimed [Weiderman 1987]. The problem is further compounded for the evaluation of development systems in which the software development paradigm violates the fundamental assumptions of long-accepted relationships between software organization and programmer productivity. Object-oriented software development environments, with all of their potential for productivity improvement, are just such systems.

Evaluation results in the research literature to date concerning object-oriented systems have been lacking in the degree to which they can be used to support further research. The intent of this proposed methodology is to systematize the evaluation of object-oriented development environments. This section identifies the design principles basic to the development of this methodology and builds on work conducted at the Software Engineering Institute, Carnegie Mellon University by identifying principles necessary to the valid evaluation of object-oriented development environments. The design principles identified in the work at CMU are insufficient to ensure a comprehensive and extensible evaluation of object-oriented development environments. The methodology design principles unique to the proposed research are so identified.

3.3.1 Based on User Activities

The performance of user software development activities is inherently sensitive to the facilities provided by any software development environment and is therefore the "common denominator" by which we can compare significantly different environments, provided, of course, that the activities of interest are accomplishable within these environments and that user performance is measured in environment-independent terms. Evaluation methodologies based on feature analyses and environment-specific performance measures become inapplicable when the environments to be compared differ significantly in the tool sets or the level of language abstraction provided to the software developer. These considerations motivate the design principle that evaluations under the presented methodology must be based on the performance of user activities.

This principle ensures a consistent basis for the evaluation of environments with widely differing support facilities. As noted in [Weiderman 1987], this is the approach adopted by Roberts and Moran [Roberts 1983] in their work on the evaluation of highly heterogeneous text editors.

3.3.2 Environment Independence

Any objective evaluation methodology, by definition, must not be inherently biased for or against the environments to be evaluated. Evaluation methodologies based upon the evaluation of particular mechanisms for accomplishing user activities rather than based upon evaluating user performance in accomplishing those activities must exhibit a bias toward environments with specific features to support those mechanisms. Evaluation methodologies based on environment-specific criteria are likewise biased toward the environments which support usage concepts closely related to those criteria, while penalizing environments with different, but possibly more efficient, mechanisms for accomplishing user development activities. For these reasons, we have adopted the design principle that the presented evaluation methodology must be environment-independent in the specification of evaluation criteria and user activities.

This principle ensures that the methodology is not biased toward any particular environment, but rather focuses on basing evaluations on user activities and formulating criteria and tests in a generic fashion.

3.3.3 Based on Experiments

The reliability of an evaluation methodology as a tool is directly related to the repeatability of the results obtained from its exercise. Results that are not repeatable are simply not convincing. The reliability of the results is also related to the degree to which the methodology produces objective evaluative results. A scientific experimental approach to evaluation directly supports both the repeatability and the objectivity of evaluative results. This is the primary motivation for including the design principle that evaluations must be based on well-defined experiments. The experiments must be conducted in rigorously defined steps with clearly defined measurements to be taken at each step.

3.3.4 Test a Core of Functionality

Software development is a highly diverse activity involving many aspects of any environment. It is therefore necessary to test a representative core of the functionality provided by the environments to be evaluated. The scope of this "core of functionality" is determined by the functionality requirements of a particular applications domain. The results of the evaluation will then indicate the degree of support for representative development activities within the selected applications domain, rather than the performance of isolated features of the environment. For this reason we have adopted the design principle that the evaluation methodology must require the testing of a core of functionality.

3.3.5 Extensible

Issues of interest in software development evolve and change with experience and technological advances. An evaluation methodology must be flexible enough to address emergent interests and capabilities within software development environments. As a result of an evaluation, attention may focus on one or more aspects of development within the candidate environments. An evaluation methodology must therefore also permit the incorporation of additional evaluation criteria and facilities. To accomplish this, the evaluation methodology must not be based on assumptions which preclude the addition of user activities, evaluation criteria or additional metrics. Existing text editor evaluation methodologies, for example, have often been based on the number of operations needed to manipulate lines of text in various ways. This basis absolutely precludes the evaluation of the new hypertext editors in which there is no concept of text lines (sentences are the basic unit of text in these editors).

To support the ability of the evaluation methodology to address new issues and areas as software technology and user interests evolve, we have adopted the design principle that the evaluation methodology must be extensible.

3.3.6 Provisions for Applications Development Paradigms

An Applications Development Paradigm (ADP) is a set of guidelines for problem decomposition and software construction. ADPs have evolved in many applications domains. In the area of user interface development, for example, development paradigms include the transition graph, hierarchic, iconic, and Model View Controller (MVC) strategies for organizing the software components of an interface. Each of these strategies is appropriate for various applications and software development environments. A consistent comparison of user performance of software development activities must account for the ADPs used in such development, since it is these development paradigms that guide the use of the software development facilities available in the development environment. Documentation and enforcement of ADPs during an evaluation also ensures that developed software represents an appropriate application of the software technology in question, thereby increasing the evaluator's confidence in a valid comparison. Consequently, the specification of ADPs is included as a design principle of the presented evaluation methodology. This principle is unique to this research.

3.3.7 Ensure the Capture of Relevant Structural Information

To fulfill the primary intent of this research to support long-term research into software complexity metrics for object-oriented systems, the evaluation methodology must provide for the capture of data characterizing the performance of users developing software, as well as data which characterizes the object-oriented aspects of that software. This data is essential to the formulation and subsequent validation of models and appropriate metrics of developer performance and product quality for object-oriented software development. The user performance data may be used independently to support the comparison of candidate environments for development activities within a specific applications domain. The structural information captured as a result of the evaluation process will be used to investigate the impacts of tools, organizational strategies, and new language features on the object-oriented aspects of developed software. This design principle is also unique to this research.

3.3.8 Automatic Primary Metric Data Capture

The reliability of any evaluation process can be compromised by the inconsistency, bias and imprecision of manually-collected performance data and manually-evaluated criteria. The cost in time and personnel of using manual data collection can make an otherwise highly desirable evaluation effort prohibitively expensive even if the manual data collection mechanisms perform flawlessly. If data is contaminated by a manual error (a much more likely outcome than errors introduced by automatic mechanisms), the entire evaluation may have to be discarded. In consideration of these issues, automatic capture of primary metric data is included as a design principle of the evaluation methodology presented as a result of this research.

3.4 A Systematic Methodology for Evaluating Object-Oriented Systems

This section presents the proposed methodology and delineates the environment-independent and environment-specific aspects of each phase. The individual phases comprising the proposed object-oriented programming environment evaluation methodology are presented in the following sections.

3.4.1 Phase 1: Identify the Applications Domain

The specific applications domain must be clearly defined and any specific areas of interest identified. This phase focuses on the specific context in which the evaluation is to be conducted. It is intended to focus consideration on the expectations of the evaluators. Test applications that are representative of the applications domain must be selected. At this time, the phase of the development life cycle under consideration must also be specified. Development phases include software development, debugging, and enhancement.

The results of this phase are the identification of the applications area, the identification of specific applications, and the identification of the software development phase of interest within the evaluation process. This phase is the first step in establishing the scope of user activities to be executed during the evaluation process and thus is supportive of the design principle that evaluations be based on performance of user activities (Section 3.3.1). The primary guideline for the execution of this phase is to select applications and development phases appropriate to the development area being investigated.

3.4.2 Phase 2: Identify the Test Development Systems

In this phase, the candidate software development systems are selected and characterized. These systems must also be evaluated for their ability to support applications of the type identified in Phase 1.

The results of this phase are the selection of the languages, the support libraries, the support tools, and the equipment platform that will comprise the evaluation environment. This phase represents a refinement of the evaluation scope from an applications area into a set of candidate development systems and thus represents the second step in establishing the scope of the user activities that form the basis of the evaluation (Section 3.3.1). The focal guideline for the execution of this phase is that the selected evaluation environments must be representative development environments employed within the applications domain identified as a result of Phase 1.

3.4.3 Phase 3: Identify the Respective Application-Specific Development Paradigms

Development paradigms relevant to both the selected development environments and to the specific test applications must be identified and/or developed in this phase. The purpose of this phase is to ensure that the environment-specific test applications are representative of their respective design strategies as well as of their respective applications domains.

The result of this phase is the identification of development paradigms for each test development system, typically one for each candidate language to be evaluated.This phase directly supports the design principle of Section 3.3.6 that requires the identification of appropriate Application Development Paradigms. The primary guideline for the execution of this phase is that these development paradigms must be selected for their ability to support software development in the applications domain of Phase 1 and for the ability of the environments selected in Phase 2 to support these paradigms.

3.4.4 Phase 4: Identify and Define Additional Metrics

Metrics of interest, but not included in the required primary metric data definitions, must be identified in this phase. Generic evaluation methods must then be defined and generic evaluation procedures developed for each of these additional metrics.

The results of this phase are the generic specification of any additional metrics and metric evaluation procedures to be supported during the evaluation process. This phase directly supports the extensibility design principle referred to in Section 3.3.5. The primary guideline for the execution this phase is to ensure that all of the characteristics of interest are captured by either the required primary metric data definitions or the additional metrics defined in this phase [Conte 1986].

3.4.5 Phase 5: Identify and Classify User Development Activities

The specific life cycle interests identified in Phase 1 must be refined into specific groups of user activities. These must be identified as developmental specifications or changes thereto in order to preserve the generic nature of these activities. It is reasonable to expect that a specificational criteria or change is implementable across the candidate development environments. These activities must comprise a representative core of the developmental functionality for the specific applications selected as a result of Phase 1.

The results of this phase are the identification of specific user development activities, the generic specification of those activities, and the generic specification of acceptance test criteria for each activity. These are very well developed software engineering issues and, as such, do not require specific explanation within this document. This phase is the third step in the refinement of the scope of the user activities for the evaluation process (Section 3.3.1) and directly supports the design principle that the user activities test a core of functionality (Section 3.3.4).

3.4.6 Phase 6: Establish Evaluative Criteria

The user performance criteria of interest must be identified and defined in this phase and strategies for automatically monitoring these criteria must be established at this time. These criteria must also be environment-independent.

The results of this phase are the generic specification of the evaluation criteria and the generic strategies for monitoring the selected criteria. This phase is supportive of the design principle that the evaluation be based on experiments (Section 3.3.3) in that it defines the terms in which the experimental hypothesis may be phrased. The primary guidelines for the execution of this phase are that the criteria should be selected from Basili's direct cost/quality criteria [Basili 1986] and that these criteria must be appropriate to the evaluative issues of interest [Shneiderman 1987].

3.4.7 Phase 7: Develop Environment-Independent Experiments

This phase encompasses the development of an appropriate experimental design and the logical integration of the monitoring facilities for the primary and additional metrics as well as performance criteria.

The primary metrics to be monitored during the experiment are intended to capture structural information relevant to object-oriented systems. This information is essential to making valid comparisons between object-oriented designs and to supporting long-term research interests in the development of formal complexity models for such systems.

Within the scope of a particular object, traditional software metrics are as relevant to object-oriented systems as they are to traditionally-developed software because, within this scope, traditional complexity factors come into play, including control, data, and temporal coupling. Traditional assumptions about the relationship between productivity and the number of lines of code are still valid. For this reason and for the ability to make comparisons between object-oriented and non-object-oriented systems, we have elected to include as primary metrics Halstead's basic software science metrics, namely, the number of unique operators, the number of unique operands, the total number of operators, and the total number of operands [Halstead 1977]. Halstead's more familiar metrics of *length, vocabulary,* and *volume* can be trivially calculated from these basic metrics and so are not included here, however, these derived metrics may correlate more closely with productivity factors under investigation. For the same reasons, we have included as a primary metric McCabe's "cyclomatic complexity" metric [McCabe 1976]. Each of these traditional (traditional in the sense that they have been used extensively on traditionally-developed software) metrics has proven to be a good indicator of software complexity and a good predictor of software development effort [JonesC 1986, Harrison 1986].

The primary metrics that pertain to the object-oriented aspects of the software under investigation are derived from the structural organization of object-oriented systems.

Objects communicate exclusively by sending and receiving messages to and from other objects. These messages can be viewed as forming edges on a graph in which each node is a unique object. To completely characterize this graph, it is both necessary and sufficient to record the destination and source of each message within an application. The pragmatics of doing this for any specific object-oriented system depend on the implementation of the messaging facility. If the messaging facility is centralized, as in SMALLTALK and Objective-C, then the monitor is likewise centralized. However, if the messaging facility is distributed in the procedure/function call mechanism, as in LOOPS, encapsulation of application objects within a monitor object is appropriate.

Another structural aspect of object-oriented systems is the inheritance lattice. This lattice represents the paths by which methods are associated with object classes. These paths are established by class references within object-oriented code. Objects may have many descendants and many ancestors in this graph, depending on the object-oriented language in question. A complete representation of this graph requires identification of all ancestors and descendants of any application object, as well as the identification of the defining class for any method, since navigation of this graph is dependent on the inheritance mechanism(s) supported by the language. This information completely characterizes the inheritance lattice and, therefore, represents another primary metric within the proposed methodology.

All metrics concerning the object-oriented structure of software under investigation can be computed from the two graphs defined above. Examples of such additional metrics include the number of messages to which an object will respond, the number of other objects to which messages will be sent, the number of parents in the inheritance lattice, the number of descendants in the inheritance lattice, the number of siblings in the objects class, and, for each method referenced within an object, the shortest path to the definition of the method in the inheritance lattice and the ratio of unique code to total code needed to support the method (degree of re-use).

On the systems level, Halstead's basic software science metrics and McCabe's cyclomatic complexity metric are inappropriate to representing the structural relationship between objects in an object-oriented system. More recent research into metrics based on structural linguistics [JonesC 1986] holds some promise, but these metrics have yet to be validated. Therefore, the primary system-level metric data to be monitored under this methodology will include the inheritance lattice and messaging graph structure. Graph sizes in terms of nodes, edges, depth, and breadth can be directly calculated from this information.

This methodology is intended to be independent of any specific experimental design.

The results of this phase are the identification of the experimental design and the logical specification of the evaluation process. This phase is pivotal in supporting the design principles that require environment-independence (Section 3.3.2), experimental evaluation basis (Section 3.3.3) and the capture of relevant structural information (Section 3.3.7). The primary guideline for the execution of this phase is that specific aspects of sound experimental design must be adhered to faithfully to ensure validity [Basili 1981a, Boehm 1981, Conte 1986, McCabe 1976, Shneiderman 1987, Soloway 1984, Weissman 1974].

3.4.8 Phase 8: Prepare the Respective Environments

In this phase, the support facilities such as graphics libraries, communications libraries, and distributed peripheral resources should be brought to comparable levels within the respective environments. This may include providing object-oriented interfaces to existing support libraries and facilities in order to provide an appropriate and meaningful comparison.

The results of this phase are the complete test development environments. This phase represents a normalization of the environments to be evaluated and is directly supportive of the environment independence design principle referred to in Section 3.3.2 in that it reduces any bias that might be caused by differences in support libraries and peripheral resource access. The primary guideline for the execution of this phase is that the resultant environments must be representative of support facilities available in the area of interest.

3.4.9 Phase 9: Develop Environment-Specific Experiments

In this phase, the environment-independent experiment is translated into the specific test environments and the monitoring facilities for metrics and performance criteria are implemented. Any clarification of the development specifications due to the specific environments must be identified and implemented here.

The results of this phase are the environment-specific monitoring facilities and the environment-specific experimental procedures. This phase is supportive of the design principles that this methodology must have an experimental basis (Section 3.3.3) and that automatic primary data capture be provided (Section 3.3.8). The primary guidelines for the execution of this phase are to ensure that the monitoring facilities are reliable and that the activity specifications are unambiguous and do completely specify the user activities to be performed during the evaluation.

3.4.10 Phase 10: Execute Environment-Specific Experiments

This phase represents the conduct of the experiment within specific environments. This phase produces the data that will be th basis of both comparative evaluations and long-term metrics research.

The results of this phase include user performance criteria measures, metrics data, and interaction logs, all of which are to be collected automatically during the course of the experiment. This phase supports directly the design principles that require the experimental basis (Section 3.3.3) and the automatic capture of primary metric data (Section 3.3.8). The primary guidelines for the execution of this phase are to follow sound experimental procedure and to verify that the collected data is reliable and complete.

3.4.11 Phase 11: Analyze Results

Once the experiment has been conducted in each of the test environments, a comparison of the user performance results can be made. The validity of this comparison and any conclusions will, of course, be dependent upon the quality of the experimental design. At this point, we also have a comprehensive characterization of the object-oriented software produced as a result of the experiment. This data can then be used in conjunction with the user performance measurements in order to test hypotheses about relationships between the data and software complexity.

The results of this phase are the hypothesis test results. This phase primarily supports the design principle that the evaluations be based on experiments (Section 3.3.3) in that valid interpretation of experimental results complete the evaluation process. The primary guideline for the execution of this phase is to carefully select the appropriate statistical tests and techniques [Conte 1986, Basili 1986].

3.5 Evaluation Methodology Summary

This chapter has overviewed existing approaches to programming environment evaluation and has identified their respective weaknesses with regard to the ability of those methodologies to support the evaluation of object-oriented environments and to support object-oriented software complexity research. Design principles for a new evaluation methodology which address these weaknesses were then identified. Finally, the specific phases of this new methodology were presented. The next chapter will demonstrate the application of this methodology to a specific evaluative situation.

Table 3.1 summarizes the relationship between the specific phases of the evaluation methodology and the methodology design principles established in Section 3.3. Each phase of the evaluation methodology supports one or more of the methodology design principles. These design principles were motivated by the criticism of existing programming environment evaluation methodologies in Section 3.2.

Table 3.2 summarizes the results of each of the evaluative phases of the evaluation methodology and delineates the language-independent, language-specific and environment-specific nature of those results. The language-independent results apply to all of the candidate languages being evaluated. The language-specific results are tailored to a specific language, but are independent of the specific implementation of that language. The environment-specific results are tailored to the specific environment used to support a candidate language.

Figure 3.1 overviews the relationship between the proposed methodology, the processes that define the environment-specific context of the evaluation, post evaluation analysis and evaluation of performance data and primary metrics, and long-term complexity model research goals of the development and validation of cognitive models for software development in object-oriented systems. These goals are motivated by the cognitive impact of the differences between object-oriented and traditional development systems. Currently defined issues for this long-term research include modeling and comparing the impact of various inheritance mechanisms and various inheritance lattice organizational strategies. Inheritance lattice characteristics have been observed to affect the time needed to become productive in object-oriented environments and the level of reusability typically achieved in such systems [Cox 1986]. The methodology presented in this chapter is critical to investigating these types of issues.

Evaluation Methodology Phase/Design Guidelines Mapping	
Phase	Design Guidelines
1. Identify applications domain.	Based on user activities (Section 3.3.1).
2. Identify test development systems.	Based on user activities (Section 3.3.1).
3. Identify applications development paradigms.	Provisions for Applications Development Paradigms (Section 3.3.6).
4. Identify and define additional metrics.	Extensible (Section 3.3.5).
5. Identify and classify user development activities.	Based on user activities (Section 3.3.1). Test a core of functionality (Section 3.3.4).
6. Establish evaluative criteria.	Based on experiments (Section 3.3.3).
7. Develop environment-independent experiments.	Environment independence (Section 3.3.2). Based on experiments (Section 3.3.3). Ensure the capture of relevant structural information (Section 3.3.7).
8. Prepare the respective environments.	Environment independence (Section 3.3.2).
9. Develop environment-specific experiments.	Based on experiments (Section 3.3.3). Automatic Primary Metric Data Capture (Section 3.3.8).
10. Execute environment-specific experiments.	Based on experiments (Section 3.3.3). Automatic Primary Metric Data Capture (Section 3.3.8).
11. Analyze Results	Based on experiments (Section 3.3.3).

Table 3.1

Evaluation Methodology Phase/Design Guidelines Mapping

Evaluation Methodology Phase/Results Mapping			
Phase	Language Independent Results	Language Specific Results	Environment Specific Results
1. Identify applications domain.	Identification of applications area. Identification of development phase. Identification of specific applications.		
2. Identify test development systems.		Identification of languages to be evaluated. Identification of library requirements. Identification of tool requirements.	Identification of hardware evaluation platform.
3. Identify applications development paradigms.		Identification of application-specific paradigms.	
4. Identify and define additional metrics.	Identification of additional metrics. Identification of generic metric evaluation procedures.		
5. Identify and classify user development activities.	Identification of specific user development activities. Generic specification of user development activities.		
6. Establish evaluative criteria.	Identification of evaluative criteria. Specification of generic monitoring facilities.		
7. Develop environment-independent experiments.	Experimental design.	Generic specification of the evaluation process.	
8. Prepare the respective environments.			Complete development environments.
9. Develop environment-specific experiments.			Environment-specific monitoring facilities. Environment-specific experimental procedures.
10. Execute environment-specific experiments.			User performance data. Software analysis data.
11. Analyze Results			Hypothesis test results.

Table 3.2

Evaluation Methodology Phase/Results Mapping

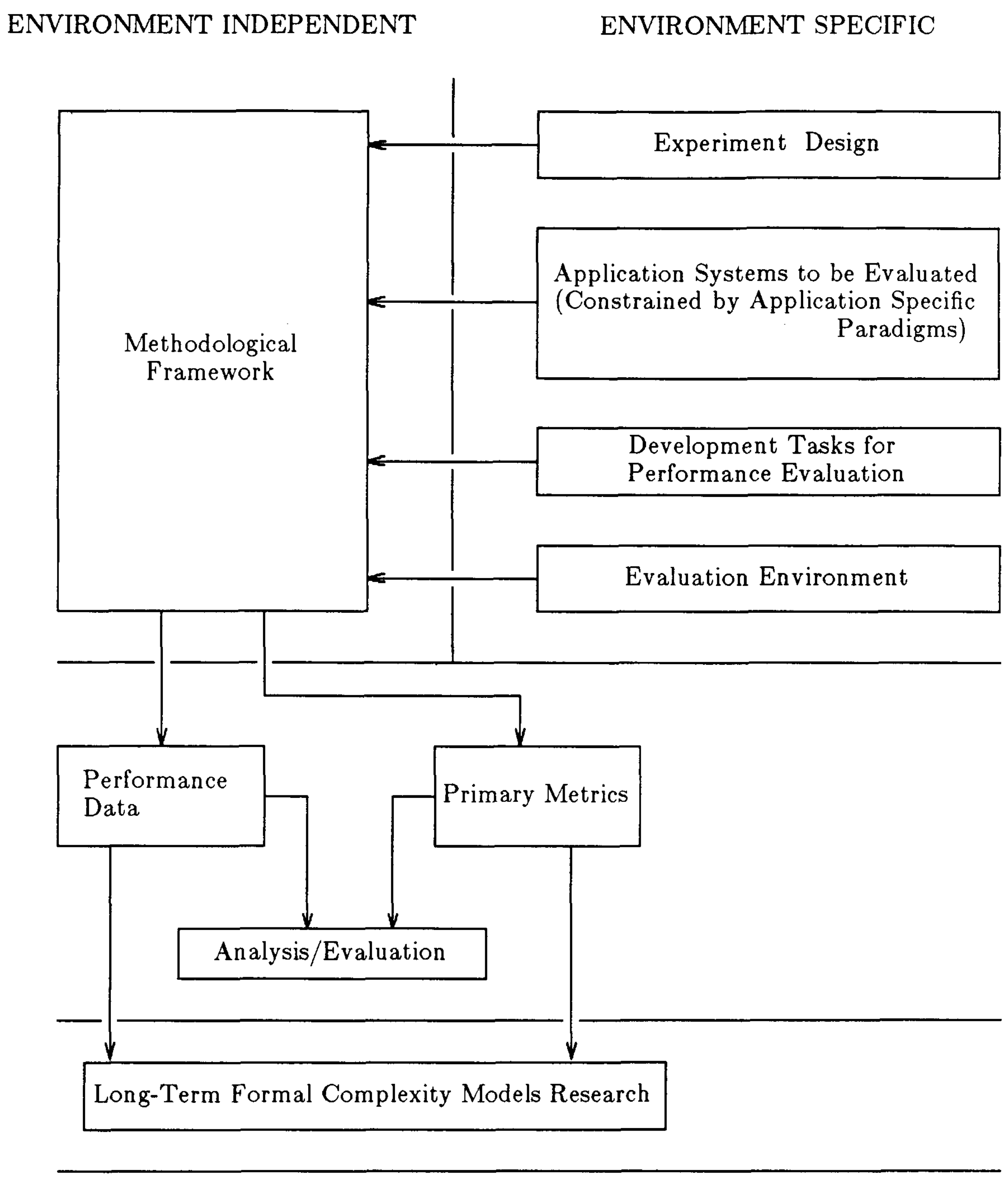

ENVIRONMENT INDEPENDENT
ENVIRONMENT SPECIFIC
Methodological Framework
Experiment Design
Application Systems to be Evaluated (Constrained by Application Specific Paradigms)
Development Tasks for Performance Evaluation
Evaluation Environment
Performance Data
Primary Metrics
Analysis/Evaluation
Long-Term Formal Complexity Models Research

Figure 3.1

An Overview of the Evaluation Methodology

CHAPTER 4: SPECIFIC TEST CASE

This chapter represents the application of the programming environment evaluation methodology proposed in Chapter 3 to a specific applications domain and set of applications development environments. It intended as a concrete demonstration of the capabilities of this evaluation strategy.

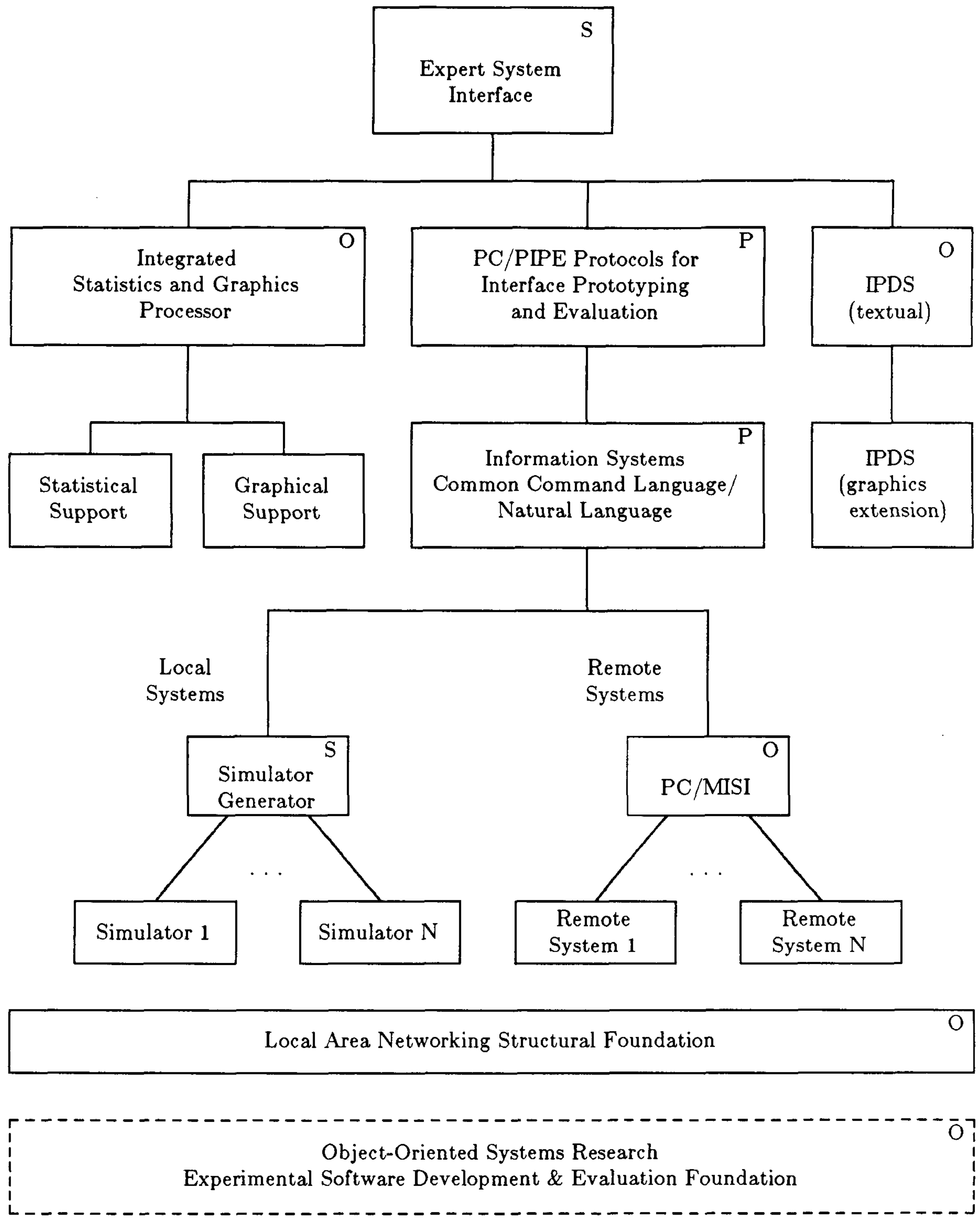
This research has grown out of a long-standing interest, within the USL NASA Project [Dominick 1987], in object-oriented systems and in the impact of this technology on traditionally difficult applications development domains, specifically those of interactive graphical applications.

As illustrated in Figure 4.1, object-oriented systems technology has now become the primary experimental software development and evaluation foundation for future PC workstation components of the USL NASA Project.

Due to our interest and expertise in the interactive graphical applications domain, we have selected this area as the domain of the specific test case presented in this chapter. This demonstration will follow the phase sequence presented in Chapter 3.

4.1 Phase 1: Identify the Applications Domain

As stated above, the applications domain identified for this specific test case was that of interactive graphical applications; more specifically, systems which present an integrated and highly interactive user interface to an underlying application. A very large fraction of the development effort required for applications in this domain is typically spent on the user interface [Cox 1986, Goldberg 1983]. Consequently, the specific interest for this evaluation was the


S
Expert System Interface
O
Integrated Statistics and Graphics Processor
P
PC/PIPE Protocols for Interface Prototyping and Evaluation
O
IPDS (textual)
Statistical Support
Graphical Support
P
Information Systems Common Command Language/ Natural Language
IPDS (graphics extension)
Local Systems
Remote Systems
S
Simulator Generator
O
PC/MISI
Simulator 1
Simulator N
Remote System 1
Remote System N
O
Local Area Networking Structural Foundation
O
Object-Oriented Systems Research Experimental Software Development & Evaluation Foundation

PROJECT STATUS LEGEND: S : Specification Stage O : Operational Stage
P : Prototyping Stage

Figure 4.1

USL NASA/RECON & NASA/JPL PC Research Projects impact of object-oriented systems design on the development of the graphical user interface.

The most general form of a graphical user interface is that of an interactive graphics editor, in that it encompasses a very broad and representative set of user interactions, including creation, selection, and manipulation of graphical objects in a variety of ways. Graphical attributes may be assigned to objects by simple icon selection. Objects are constructed, placed, and manipulated using combinations of selection and location interactions. Due to its generality and importance to graphical applications, the specific application identified for this evaluation was an interactive graphics editor.

In this evaluation, the development phase identified was the product maintenance/enhancement phase [Balzer 1986], more precisely, the addition of specific capabilities to the graphics editor application. Activities in this phase of development account for most of the cost of a software system over its lifetime [Cox 1986, JonesC 1986].

4.2 Phase 2: Identify the Test Development Systems

Since the focus of this evaluation was the impact of object-oriented technology with respect to traditional graphical applications technology on interactive graphical applications development, it was appropriate to identify both an object-oriented and non-object-oriented development system. Other evaluations that focus on the impact of specific object-oriented features, inheritance mechanisms for instance, would identify multiple appropriate object-oriented development systems.

The development languages identified for this evaluation were C and AT&T's object-oriented extension to C, namely, C++ [Stroustrup 1986]. Each of these languages is based on the UNIX operating system and supports standard C calling sequences to support libraries. The complete availability, from the USL NASA Project, of seven identical networked AT&T 7300 UNIX PCs with mice led to the selection of this workstation as the platform for this evaluation. The fact that these workstations, under the control of the USL NASA Project, could be completely dedicated to evaluation activities simplified many configuration, monitoring, integration, and scheduling issues. The primary software interface to the graphical capabilities of this workstation was through the AT&T Virtual Device Interface (VDI) and Window Control (WC) libraries. A C++ interface to these libraries was needed to preserve consistency in the C++ development tasks. Only the standard UNIX development tools *make* and *vi* were used to support development activities since they were readily available and were equally applicable to both the C and C++ environments. The *make* facility also insulates users from having to deal with complicated compiler/linker command line syntax and library/object module interdependencies. The facilities provided by these workstations and these graphical libraries are highly typical of those provided within commercial graphical applications development environments.

4.3 Phase 3: Identify the Respective Application-Specific Development Paradigms

Application-specific development paradigms are intended to ensure that the applications developed using the environments under evaluation are representative of the respective software development technologies. In this evaluation, a specific paradigm was identified for each development environment. Other evaluations considering only object-oriented systems may require only one applications development paradigm.

4.3.1 C Environment Applications-Specific Paradigm

For the C-based environment, the applications paradigm was the traditional Graphical Kernel System (GKS) applications organization strategy. In this development paradigm the application is organized into functional modules representing the individual graphical interaction capabilities. These functional modules are then imbedded in a nested "case" statement control structure representing possible interaction sequences. Additional capabilities are added by providing any new interaction capabilities (e.g., line style selection) and modifying the control structure appropriately. This paradigm is taught (however implicitly) in virtually every introductory graphics class and text book.

4.3.2 C++ Environment Application Design Constraints

For the C++-based environment, the application-specific development paradigm has been developed as part of this research. For software to be appropriately representative of object-oriented development, it must exhibit evidence of the use of object-oriented techniques in its structural organization. Object-oriented software typically exhibits a high degree of re-use of internal methods and a large number of small objects that communicate in patterns similar to the structure of the application being modeled. The aspects of structural organization unique to object-oriented systems involve the inheritance lattice, representing the relationship between methods and objects, and the messaging graph, representing the relationship between objects and other objects. Maximizing the re-use of methods defined throughout the inheritance lattice requires that these methods be appropriately factored into common parent object classes. Methods that are used in multiple objects but not factored into the inheritance lattice are clearly replicated and, so, not re-used. Additionally, to maximize the generality and, thereby, the utility of any individual object, the object should be constructed to deal with a small set of messages and to manipulate a single principal data structure. The more messages and data structures an object manipulates, the more specialized and less generic is its function.

The fundamental design constraints that affect to what degree objects make use of object-oriented facilities are the degree to which methods are factored into the inheritance lattice and the degree to which an object focuses on a specific function or capability.

4.3.3 C++ Applications-Specific Paradigm

The applications-specific paradigm developed pursuant to these design constraints is presented in this section.

Each graphical entity must be represented as a separately instantiated object with the appropriate methods for reporting its characteristics in response to generic messages. Typical methods include current object location, size, creation, deletion, selection and display operations. Private object data structures must be sufficient to support local methods. Common methods and data structures must be factored into classes to be used in constructing the inheritance lattice.

Objects are to be organized into two disjoint groups, namely, user interface objects and generic application objects. Graphical interactions must occur through the user interface objects which must, in turn, activate appropriate generic objects to accomplish non-interface oriented functions. The intent of this partitioning of objects is to isolate graphical interaction functions from applications-specific functions like file access and generic data manipulation. The term "generic" in this context refers to functions which may be activated by a variety of interaction mechanisms and that are not directly associated with support of the user interface.

This applications paradigm enforces decomposition of the interactive graphical application into objects which are directly related to the entities with which the user interacts and limits the scope of such objects to one such entity. This forces any interaction between graphical entities to be accomplished through the messaging mechanism, preserving the extensible and re-usable nature of these objects. Additionally, the factoring of common methods into the inheritance lattice preserves another important characteristic of object-oriented systems without constraining just which methods are implemented. The user interface structure of any interactive graphical application developed under this paradigm will be representative of an appropriate application of object-oriented techniques.

4.4 Phase 4: Identify and Define Additional Metrics

The methodology presented in Chapter 3 does not require identifying and defining additional metrics, but certainly does not preclude doing so. This would, of course, require the implementation of additional monitoring facilities. For this evaluation, no metrics other than those required by the methodology were identified.

4.5 Identify and Classify User Development Activities

Three independent development activities were identified for this evaluation. These tasks were selected to be representative of commercial software maintenance/enhancement activities. By far the most predominant features available in commercial graphics editors are the large variety of geometric primitives. The set of primitives available to the user of a graphics editor directly affects that user's productivity since he must himself build anything not directly available within the editor. The primary difficulty, for the developer, in implementing new geometric primitives lies in the integration of that primitive into the icon/menu system that comprises the user interface. The implementation of geometric primitives are highly representative of commercial graphics editor development activities in both functionality and difficulty. For this reason, the first task identified was the addition of a rectangle geometric primitive to the base graphics editor. The second development task identified was the implementation of another geometric primitive to permit opportunities for code re-use and attribute *polymorphism.* Polymorphism, in this case, refers to the implicit selection of appropriate implementations of an attribute based upon the type of the geometric primitive with which it is associated. Finally, the development of an attribute primitive was identified as the third task. Attributes are the next most prevalent feature in commercial graphics editors and include color, fill, texture, selectability, blinking, and intensity. The difficulty involved in integrating the attribute primitives is greater than that of geometric primitives due to the need for establishing an additional level of icons/menus. An attribute primitive is also a highly representative feature and thus its implementation is a highly representative development task. We have selected the implementation of a solid fill attribute for this third and final task.

As stated above, the first development task identified was the addition of a rectangle geometric primitive to the base graphics editor. The rectangle was to be constructed by selecting the appropriate menu item with the mouse and subsequently selecting two opposing diagonal corners indicating the size and position of the new object. Certification of correct completion of this task involved testing the new capability over its input equivalence classes. For the rectangle, primitive this required constructing the rectangle using the four possible combinations of opposing diagonal corners, namely, upper right to lower left, lower left to upper right, upper left to lower right, and lower right to upper left. Testing of the existing line drawing primitive was also required to ensure that it had not been damaged as a result of development task activities.

The second development task identified was the addition of a triangle geometric primitive to the base graphics editor. The triangle was to be constructed by selecting the triangle menu item and subsequently, the three vertices of the triangle. Task certification for this primitive involved testing over only two input equivalence classes, namely, the clockwise and counter-clockwise entry of the triangle vertices. Again, testing of the existing primitives was required to ensure that no damage had occurred.

The final development task identified was the addition of solid fill capability to both the rectangle and triangle geometric primitives. Once the geometric primitive was selected, an attribute menu was to appear, allowing the selection of a filled or hollow attribute for that primitive. Certification of this task required testing both the rectangle and triangle geometric primitives as specified above for both filled and hollow attributes. Testing of the existing line-drawing primitive was required once again to ensure its continued correct operation.

The development tasks as specified in this section are completely generic with respect to the candidate development environments.

4.6 Phase 6: Establish Evaluative Criteria

The proposed methodology is independent of the specific criteria used to evaluate subject performance on the user development tasks identified as a result of the Phase 5 activities, as long as the criteria are environment-independent in definition. Potential criteria are based on Basili's direct cost/quality criteria [Basili 1986] and include measures of cost, errors, changes, and reliability.

This test case application of the proposed evaluation methodology identified total development time as the primary performance criteria. Total development time was defined as the time from the delivery of task specification to the successful completion of the certification tests for each task. As noted above, this choice of criteria, being environment-independent, does not affect the validity of the proposed evaluation methodology. Time-stamped interaction transcripts were identified as the generic facilities for monitoring this criteria, since this approach permits a detailed characterization of each subject's activities during the experiment. For example, we can easily derive the number of compiles, time spent compiling, number of editor invocations and time spent editing during a development session from interaction transcripts by classifying the time-stamped interactions as compile-related or edit-related and accumulating the respective interaction times and counts. Other evaluations might include other performance criteria. The identification and documentation of such criteria is critical to ensuring the comparability of the results of independent evaluations.

4.7 Phase 7: Develop Environment-Independent Experiments

This methodology is also independent of the specific experiment design identified in support of the evaluation. The specific hypotheses are, however, constrained to assertions involving environment-independent performance criteria monitored during the conduct of the identified development tasks. The data collected during the experiment is also constrained to include the primary metrics identified in Section 3.4.7. Selection of the specific experiment design must be based upon standard experimental considerations including the number of subjects available, the time commitment of those subjects, the degree to which the population is homogeneous, the number of treatments involved, and the degree of control desired for "maturation" effects [Conte 1986]. Since these considerations are specific to the environment in which the evaluation is to be conducted, the proposed evaluation methodology must assume that the experiment design is appropriate and that the experiment is conducted according to sound experimentation procedures. The validity of the proposed methodology in no way depends on the validity of the design or conduct of any specific experiment.

The null hypothesis identified for this experiment was that there would be no difference between the paired development time distributions for development in the C and C++ environments. Determination of the level of significance of the results of this experiment required computation of the probability of wrongly rejecting this hypothesis.

The experiment population for this evaluation consisted of Computer Science seniors and graduate students who had extensive C, UNIX, and graphics experience, in particular, who had completed major course-related projects (e.g., operating systems, databases, etc.) using C in a UNIX environment and who had experience using at least two highly-interactive graphical tools (e.g., paint packages, drafting packages, graph editors, etc.). Four subjects participated in the experiment whose activities spanned two months. Since multiple observations were to be conducted involving related tasks, there was a need to control for maturation effects. This consideration led to the identification of a counter-balanced design for the experiment.

The generic evaluation procedure specified for this test case consisted of five steps, as follows:

(1) The subjects are to be randomly assigned to one of two groups. The first group is to approach the development tasks in C on the first day and in C++ on the second day, while the second group is to approach the development tasks in C++ on the first day and in C on the second day.

(2) C++ and VDI proficiency are to be tested immediately before beginning the first development task on each of the two days.

(3) Each development task is to be uninterrupted and begins with the disclosure of task specifications and ends with successful certification of the task.

(4) Upon completion of the first task, the subjects are to logout of the workstation environments (this permits flushing all transfer buffers and re-initialization of storage management facilities to protect data in the

event of a power outage; this is the only way to re-initialize the 7300's virtual memory management mechanisms to prevent swap space fragmentation and the associated excessive performance degradation).

(5) The next development task is to be started immediately upon completion of the previous task until all three tasks for the day are completed. A maximum time limit of four hours is specified since development activities exceeding this limit would only occur in the pathological case of the subject chronically overlooking a particular error, inappropriately affecting the observed development time distributions. In such a case, the respective data would be considered missing.

Primary focus was to be on the differences in development time in C versus the development time in C++ for each subject. This experimental organization is similar to that used by Gannon [Gannon 1977] in evaluating the impact of strong typing on program development.

4.8 Phase 8: Prepare the Respective Environments

The C and C++ environments within the selected NASA AT&T workstation environments were identical except for the Virtual Device Interface (VDI) and Window Control (WC) mechanisms. A C++ language interface mechanism was developed in this phase to provide C++ users access to VDI and WC at the same level of abstraction and functionality as was available to the C users. These libraries were tested by using them to construct the base editor applications. The intent here was to provide support function invocation in a style consistent with C++.

The graphics editor applications were also developed in this phase. These editors provided the software base to be used in the development activities and were extensively tested for equivalent functionality and reliability by exercising all primitives over identified input equivalence classes. No functional differences existed between the environments in either the available tools or the base graphics editors.

4.9 Phase 9: Develop Environment-Specific Experiments

Automatic performance monitoring facilities for the performance criteria established in Section 4.6 were implemented in both the C and C++ environments. These facilities were completely transparent to the user and consisted of an interaction monitor imposed between the user and the operating system command shell. Each interaction is captured and time-stamped for start and finish times. The interaction is then passed on to the standard shell for execution. This data is collected in log files that are distinct for each development task. A file name generation scheme was developed to avoid filename collision. Automatic documentation facilities for subject, time, and workstation identification was also implemented as part of this phase.

Automatic primary metric computation facilities for C and C++ were based on LEX LR(1) grammars. The grammars essentially support the counting of operands, operators and conditional statements in support of the traditional metrics. To determine the inheritance lattice in C++ software, it is only necessary to capture the names of all of the members defined in all of the classes of the application. This was accomplished by automatic analysis of the class definition sections of the software, identified by the grammar, and the construction of class member tables. Determination of the messaging graph, however, required runtime information, since the binding of messages to C++ *virtual functions* occurs during execution (virtual functions are a specific type of C++ method). To support construction of the messaging graph, class member tables were augmented to capture argument typing for each defined method and static class identifier strings were imbedded within class declarations by establishing standard class header macros which are expanded during compilation.

These facilities for providing automatic primary metric computation were completely implemented with the exception of the C++ messaging graph facility. The messaging graph data collection mechanism was completely designed but was not implemented due to vendor delays in the delivery of the C++ translator source code. The source code for the translator was needed to gain access to internal runtime messaging mechanisms known as *vtables*. A compromised mechanism could have been implemented using static code analysis techniques alone; however, this static approach was foregone in favor of the technique described above. The implemented primary metric computation facilities were tested on the National Institute of Health's OOPS C++ library [Gorlen 1986] and the equivalent generated C code and performed correctly.

Environment specific preparation of the subjects was also conducted in this phase. The subjects involved in the experiment were extensively trained in C++ over a two month period. Training effectiveness was monitored by testing subjects on the C++ features and concepts incorporated within the base graphics editors. The subjects were also required to complete programming exercises on the AT&T 7300 workstations using C, C++ and the Virtual Device Interface libraries.

The generic experimental procedure of Phase 7 required no refinements for the specific environments used in this evaluation and was adopted as the environment-specific experimental procedure with no changes. In evaluations within which the facilities for the candidate environments differ appreciably, this phase would include distinct environment-specific experimental procedures.

4.10 Phase 10: Execute Environment-Specific Experiments

This phase represents the actual conduct of the environment-specific experimental procedure developed in Phase 9.

The four subjects were randomly assigned to one of two experimental groups. The first group, containing subjects 1 and 4, approached development in C++ first and subsequently in C. The second group, containing subjects 2 and 3, approached development in the opposite order.

Before each set of development tasks, the subjects were tested for proficiency in both C++, VDI, and WC. Tables 4.1 and 4.2 summarize the results of these tests for both sets of development tasks. The tests covered all of the features and concepts incorporated within the base graphics editors that were not a part of standard C programming. The maximum possible score on each test was 12. The minimum threshold for participating in the experiment was established to be 80% correct, or 10 out of 12. As can be seen from the tables, all subjects met or exceeded this threshold for both days of the evaluation without additional training.

The specifications were distributed to all subjects at the same time. A maximum time limit of four hours was set for each development set. The development tasks proceeded without interruption until the entire set of three development tasks was completed and the certification tests passed. On the following day, the groups were reversed and re-tested for proficiency. The subjects were then allowed to proceed with the development task sets. The user performance data gathered (in real time seconds for task completion time) is presented in Tables 4.3 and 4.4, and represents the total development times for each task for each subject as monitored during the experiment. The characteristics of the base graphics editors and the completed editors from each of the subjects in both C and C++ are summarized in Tables 4.5, 4.6 and 4.7.

4.11 Phase 11: Analyze Results

As stated in Phase 7, the null hypothesis for this experiment was that there is no difference between the development time distributions in C and C++. Since sample sizes were small and sensitive to isolated bad performances, nonparametric statistical tests were in order, even with the greater risk of accepting a false null hypothesis [Basili 1986]. The performances of each of the subjects in C and C++ were compared using a one-tailed Wilcoxon test [Gannon 1977] since these are related measures. From the analysis of the data in Tables 4.3 and 4.4, the performance data exhibited 11 samples with a positive difference averaging rank 6, and 1 sample with a negative difference at rank 11. The calculated probability of wrongly rejecting the null hypothesis is .07 for this data, just over the standard .05 level [Conte 1986] (.1 is often used in software productivity studies [JonesC 1986, Basili 1981b, Conte 1986], although most modern experimenters do not adhere to a rigid significance threshold [Conte 1986]). We consider this level significant and we conclude that there is a significant difference between the development time distributions in C and C++ for the context of this experiment. Further, the C++ development time was often half of the C development time for these development tasks as shown in Tables 4.3 and 4.4. In particular, subjects 2, 3 and 4 exhibited total development times (combining times for all three tasks) in C++ that were 64, 50 and 51 percent of those in C respectively, while for subject 1 C++ development took 45 percent of the time required for equivalent C development for tasks 1 and 2 combined. Subject 1 encountered difficulty in debugging during task 3 in C++, accounting for the inordinate time consumed.

As initially stated in Section 4.6, as a framework for object-oriented programming environment evaluation, the validity of the methodology developed as part of this research does not depend on the outcome of any specific evaluation or on the validity of any supportive experiments. The purpose of this specific test case application of the developed evaluation methodology is solely to provide a concrete demonstration of its systematic nature and evaluative capabilities.

The VDI and C++ proficiency tests were administered prior each of the development sets to permit the identification of any "learning" effects. Table 4.1 presents the results of the VDI and C++ proficiency tests administered prior to the subjects beginning the development sets on the first day of the experiment. All subjects' scores exceeded the 80% threshold needed to participate in the experiment. Table 4.2 presents the results of the same tests administered on the second day of the experiment. Once again, all subjects' scores exceeded the minimum threshold. The distributions of these scores did not differ significantly between testings, indicating the absence of any significant "learning" effect. The administered tests are contained in Appendix B.

Table 4.3 presents the total development times, measured in real-time seconds, for each subject by tasks for development in C. Table 4.4 presents the corresponding development times for development in C++. The development time for Subject 1 for task 3 resulted from difficulty encountered in debugging. The complete monitor script used during this evaluation is included in Appendix C.

Table 4.5 summarizes the object-oriented characteristics of both the base graphics editor and the editors completed by each subject. The LEX code used during this evaluation for supporting this object-oriented characteristics analysis is included in Appendix D.

Tables 4.6 and 4.7 present the traditional metric characteristics of both the base graphics editor and the editors completed by each subject for C and C++ development respectively. The metrics used are McCabe's "Cyclomatic Complexity" and Halstead's basic "Software Science" metrics including N_1 (total number of operators), N_2 (total number of operands), n_1 (number of unique operators), n_2 (number of unique operands), N (total number of tokens (size)), n ($n_1 + n_2$ (vocabulary)), and V ($N * \log_2(n)$ (volume)). The LEX code used during this evaluation for traditional metrics analysis is included in Appendix E.

The large disparity between the measured values of the traditional metrics for the C and C++ graphics applications is attributable to the explicit typing, inter-object messages, and method declarations of C++, in contrast to defaults, side-effects, and simple function definitions that were employed in C. Additionally, as can be seen from Table 4.5 subjects added an average of three classes, containing of average of four members during development, facilitating very localized (primarily intra-class) code modification. The C++ mechanisms, while more verbose, are seen as central to improving productivity by many software engineers [JonesC 1986, Cox 1986]. This view is consistent with the results achieved in the test case PEEM execution. The ability to formally validate this sort of intuition is the principle motivation of this research.

Proficiency Test 1 Scores				
	Subject 1	Subject 2	Subject 3	Subject 4
VDI	11	12	12	12
C++	12	12	11	10

Proficiency Test Scores Prior to the First Development Set
Table 4.1

Proficiency Test 2 Scores				
	Subject 1	Subject 2	Subject 3	Subject 4
VDI	12	12	12	12
C++	12	11	12	11

Proficiency Test Scores Prior to the Second Development Set
Table 4.2

User Performance in C (seconds)				
	Subject 1	Subject 2	Subject 3	Subject 4
Task 1	754	643	378	992
Task 2	425	331	225	306
Task 3	1101	1263	727	1461

Task Completion Times for C Development
Table 4.3

User Performance in C++ (seconds)				
	Subject 1	Subject 2	Subject 3	Subject 4
Task 1	379	515	150	635
Task 2	150	219	119	161
Task 3	2675	702	405	616

Task Completion Times for C++ Development
Table 4.4

Object-Oriented Characteristics Summary					
	Subject 1	Subject 2	Subject 3	Subject 4	Base
Total Classes	8	8	9	8	5
Avg. Members/Class	4	4	3.78	4.25	5
Avg. Lines/Class	9.63	9.88	9.56	9.88	10.6

Summarized Object-Oriented Characteristics of C++ Graphics Applications
Table 4.5

Traditional Metrics for C-Based Code					
	Subject 1	Subject 2	Subject 3	Subject 4	Base
McCabe	19	13	13	15	9
N_1	749	748	707	683	440
N_2	535	565	515	458	305
n_1	31	32	31	29	29
n_2	113	132	133	122	96
N	1284	1313	1222	1141	745
n	144	164	164	151	125
V	9206	9660	8991	8259	5190

Summarized Traditional Metrics for C-Based Graphics Applications
Table 4.6

Traditional Metrics for C++-Based Code					
	Subject 1	Subject 2	Subject 3	Subject 4	Base
McCabe	33	28	27	29	19
N_1	1149	1120	1057	1086	776
N_2	820	802	745	812	541
n_1	53	52	51	53	52
n_2	198	210	200	211	181
N	1969	1922	1802	1898	1317
n	251	262	251	264	233
V	15695	15440	14365	15268	10357

Summarized Traditional Metrics for C++-Based Graphics Applications
Table 4.7

CHAPTER 5: EVALUATION OF RESEARCH WITH RESPECT TO RESEARCH OBJECTIVES

The specific research objectives which have served as guidelines for the development of this Programming Environment Evaluation Methodology for Object-Oriented Systems were identified in Chapter 2. These specific objectives were grouped under the following three general research objectives:

1. The design, development, application, and evaluation of a systematic, extensible, and environment-independent evaluation methodology capable of supporting investigation into the impact of object-oriented design strategies on the software development process (Section 2.1).

2. The design, development, application, and verification of domain-specific applications development paradigms to support consistent comparisons of applications developed under an object-oriented strategy (Section 2.2).

3. The design, development, application, and completeness verification of primary metric data definitions appropriate to systems developed under an object-oriented design strategy (Section 2.3).

This research will be evaluated according to the extent to which it has attained the specific research objectives stated in support of each general research objective and, subsequently, according to the extent to which it has satisfied the intent of that general research objective. The significance of each of the research objectives was already identified in Chapter 2. Table 5.1 presents a mapping of the research objectives to their associated evaluative sections.

Research Objective/Evaluation Section Mapping	
Research Objective	**Section**
Evaluation Methodology General Research Objective	Section 5.1
Associated Specific Theoretical Objective: To develop design principles that ensure a systematic, reproducible, and environment-independent PEEM.	Section 5.1.1
Associated Specific Methodological Objective: To design a PEEM that incorporates automatic performance and primary metric data capture.	Section 5.1.2
Associated Specific Developmental Objective: To develop a prototype evaluation environment capable of supporting the PEEM.	Section 5.1.3
Associated Specific Evaluative Objective: To conduct a demonstrative test case execution of the PEEM comparing development in a traditional and an object-oriented language.	Section 5.1.4
Application-Specific Paradigm General Research Objective	Section 5.2
Associated Specific Theoretical Objective: To determine application domain-specific design characteristics that promote consistent application of object-oriented technology.	Section 5.2.1
Associated Specific Methodological Objective: To design application domain-specific development paradigms that support effective application of object-oriented design techniques.	Section 5.2.2
Associated Specific Developmental Objective: To develop procedures for the application of the applications domain-specific paradigms within a specific object-oriented environment.	Section 5.2.3
Associated Specific Evaluative Objective: To verify that the application-specific development paradigm does produce representative object-oriented software.	Section 5.2.4
Primary Metrics General Research Objective	Section 5.3
Associated Specific Theoretical Objective: To design primary metric data definitions that characterize the aspects of software unique to object-oriented designs.	Section 5.3.1
Associated Specific Methodological Objective: To develop language-independent methods for capturing primary metric data for object-oriented designs.	Section 5.3.2
Associated Specific Developmental Objective: To provide language-specific acquisition of the primary metric data for the object-oriented systems under evaluation.	Section 5.3.3
Associated Specific Evaluative Objective: To comprehensively test these metric data acquisition methods for accuracy.	Section 5.3.4

Table 5.1

Research Objective/Evaluation Section Mapping

5.1 Evaluation of the Evaluation Methodology General Research Objective

This section presents the evaluation of the evaluation methodology general research objective identified in Section 2.1. Each specific research objective is considered individually and represents a different aspect this general research objective.

5.1.1 Evaluation of the Associated Specific Theoretical Objective

To develop design principles for the proposed evaluation methodology that ensure systematic, reproducible, and environment-independent performance evaluations, thereby supporting the long-term development of theoretical models and metrics for the characterization and comparison of object-oriented systems.

The theoretical aspect of this general research objective represents the establishment of design principles for the presented PEEM (Section 3.3). These design principles were developed based on an analysis of existing PEEMs (Section 3.2) with respect to our stated research goal of supporting long-term research into object-oriented complexity models/metrics (Chapter 1). The primary motivation for, and justification of each of the PEEM design principles were presented in Section 3.3. The support of these principles for the desired systematic, reproducible, and environment-independent nature of the PEEM was also established in Section 3.3. Table 3.1 established the mapping between the design principles and the individual phases of the PEEM. What remains is to establish that the PEEM developed pursuant to these design principles is, indeed, systematic, reproducible, and environment-independent in nature.

5.1.1.1 Systematic Procedure

The systematic nature of any methodology refers to the degree to which the relationships between the individual phases of that methodology have been established. This section evaluates the systematic nature of the presented PEEM.

This PEEM provides a structural framework for the evaluation of object-oriented programming environments. The individual phases of the PEEM are defined generically, but specific results are identified and specific execution guidelines are established for each phase (Section 3.4).

The first seven phases of the PEEM embody a procedure for establishing the scope of the evaluative activities to be pursued. This procedure includes the specification of the applications domain, the selection of candidate development environments for evaluation and comparison, the development/selection of applications domain-specific development paradigms, the identification of metrics of interest, the identification of subject development activities, the establishment of evaluative criteria, and the design of generic experiments.

The results of these phases are subsequently refined into environment-specific activities and procedures that support the evaluation process. Phases 1-7 can be refined independently of the specific evaluation facilities available. The applications domain is refined into specific representative applications. Specific implementations of the selected development environments are selected, providing the base hardware and software facilities for the evaluation. The applications-specific development paradigms are refined into environment-specific development procedures. Metric definitions are refined into environment-specific computation procedures. Subject development activities are translated into environment-specific development specifications. Evaluative criteria are refined into environment-specific monitoring procedures. Generic experiment designs are translated into environment-specific experiment designs. These activities, taken together, establish the specific evaluative procedures for each environment.

Phases 8 and 9 address the normalization of specific environments and the implementation of experiment support facilities within those environments. These activities are based on the results of Phases 1-7 and require access to the specific evaluative environments for execution. These activities must also be completed before the start of Phase 10.

In Phase 10, the environment-specific experiments are conducted in each environment, with associated performance and primary metric data being collected. Clearly, this phase must be complete before before the start of Phase 11.

Finally, in Phase 11, the results of the experiment are analyzed and any formulated hypotheses are tested. At this point, we have valid experimental results, consistent and complete performance data, and well-characterized evaluative environments. We can then immediately assess performance differences in the environments being compared.

In summary, for each phase of this PEEM, this research has identified specific results and specific execution guidelines. As shown in this section, these phases have been logically grouped and inter-group execution dependencies have been determined from the identified phase results. With this information, application of the presented PEEM can be carried out in a systematic manner.

5.1.1.2 Reproducible Evaluations

The reproducible nature of any PEEM is dependent on two primary factors, namely, adequate documentation of the evaluation context and the employment of an objective evaluation mechanism. If the context in which the evaluation occurs is thoroughly documented, then an equivalent context can clearly be established (for all practical concerns). Given an equivalent context, an objective evaluation mechanism must produce statistically equivalent evaluative results. The basis of the evaluation of the reproducible nature of this PEEM is the adequacy of its provisions for documenting the context of an evaluation and the objectivity of its evaluative mechanism.

The documentation provided by this PEEM is composed of the collection of specific phase results as summarized in Table 3.2. This collection of results completely characterizes the scope addressed in the evaluation, the candidate environments evaluated, the development paradigms employed, the tools and techniques used to gather data, the experimental design employed, and the statistical techniques applied. This completely documents the process of preparing for, and conducting the evaluation. It should be noted that this level of documentation is far in excess of that required by existing and widely accepted PEEMs and certainly exceeds the information typically provided to the research community. However, in our application of this methodology, we have found this level of documentation to be both justified and workable.

The core of the evaluation mechanism employed within this methodology is the controlled experiment. As utilized within the methodology, this forces a formalization of the questions in which an evaluator is interested and discourages qualitative evaluations. This approach also ensures objectivity with respect to the formulated hypotheses [Weiderman 1987] and thereby directly supports reproducible results. The evaluation mechanism is further supported by the automatic collection of performance and primary metric data. This circumvents the inherent bias and unreliability of manual data collection mechanisms (e.g., stopwatches, manual counting, etc.).

Either of the two issues discussed in this section would, taken alone, be insufficient to ensure reproducible evaluative results, since inadequate documentation would admit uncertainty about the equivalence of evaluation contexts and the lack of an objective evaluation mechanism or a reliable data collection mechanism admits variability in the evaluation process itself. In Section 3.2, existing PEEMs were criticized for several factors affecting the reproducibility of evaluative results, namely, the adoption of subjective evaluation criteria, the selection of inherently biased evaluation criteria, and the incorporation of unreliable data collection mechanisms. The PEEM developed as a result of this research directly addresses each of these criticisms of existing PEEMs with respect to the reproducibility of evaluative results.

5.1.1.3 Environment Independence

Environment-independence, in this context, implies that the PEEM is not biased toward or away from any candidate environment. This objective is motivated by the environment-specific nature of performance criteria, metric data, and evaluative mechanisms defined within existing PEEMs, inherently limiting the applicability of these PEEMs across heterogeneous environments. Environment-independence is not intended to imply that all evaluations of all environments will be comparable under the PEEM. Clearly, any evaluation results must also depend upon the applications domain selected, the test development systems identified, the applications development paradigms employed, and the user development activities performed. Rather, environment-independence ensures that evaluations of highly heterogeneous software development environments, where other environmental factors are comparable, will be comparable. To accomplish this, the specification of the PEEM must be independent of any environment-specific requirements and the evaluative process itself must not be biased as a result of the conduct of any phase.

Observe that each phase in the PEEM is defined in generic terms, that the results of each phase are identified in generic terms, and that the execution guidelines for each phase are specified in generic terms; that is, there are no environment-specific references in the specification of the PEEM. The environment independence claim, then, depends solely on how well the PEEM preserves the environment-independence of the evaluative process.

There are only three evaluative phases within the PEEM that have a potential biasing effect on the evaluative process, namely, Phase 4, Phase 6 and Phase 7. These phases establish additional metric data definitions, performance evaluation criteria, and the experimental design, respectively. Phase 3, the identification of application-specific paradigms, is intended to promote valid comparisons across highly heterogeneous environments, but the results of this phase affect test case consistency only, and not the evaluative process. This issue is evaluated in Section 5.2. The remaining phases of the methodology affect and are affected by the evaluative context, thereby affecting evaluative results, but these phases cannot bias the evaluative process itself.

If the results of Phase 4 are defined in environment-specific terms, then the collected metric data may not be comparably defined for widely differing environments. For example, it is ridiculous to compare the number of lines of code for a specific application in assembler versus Smalltalk. The PEEM developed as part of this research requires that any additional metrics must be defined in environment-generic terms. This ensures that, for any candidate environments, collected metrics can be compared based on this environment-generic definition.

If the results of Phase 6 are defined in environment-specific terms, the evaluative criteria may not be equally applicable across heterogeneous environments. Consider the problem of comparing productivity in lines of code/month in assembler versus Smalltalk. The assembler programmer would look unduly more productive. This PEEM also requires that performance criteria must be selected from Basili's direct cost/quality criteria. Again, this ensures that criteria from widely varying environments can be compared.

Finally, if a specifics experimental design were part of the PEEM specification, the PEEM would be inherently limited to evaluative contexts in which that experimental design is appropriate. Since good experimental design is driven by the context in which the experiment is conducted (e.g., number of subjects, variability in expertise of subjects, etc.), this would be a crippling problem. The PEEM presented in this document treats experiment-related activities as independent support activities. The PEEM assumes that the experiment design is valid and that sound experimental procedure is followed, as stated in Sections 3.4.7 and Section 4.7; however, to preserve the comparability of experimental results, the PEEM does constrain hypothesis formulation to assertions concerning the environment-generic performance criteria of Phase 6. An experiment's validity depends exclusively on the appropriateness of the experimental design to the specific evaluation environment, and on the soundness of the conduct of the experiment within that environment [Conte 1986, Basili 1986]. The considerations involved in selecting and executing a specific appropriate experimental design are not within the scope of this PEEM.

Additionally, the specification of Phase 7 in Section 3.4.7 includes the definition of required primary data metrics (to be evaluated in Section 5.3) that provide a complete characterization of the object-oriented aspects of the object-oriented test applications. These required primary metric data definitions also support the comparability of experimental results, but cannot bias the evaluative process.

5.1.2 Evaluation of the Associated Specific Methodological Objective

To design an evaluation methodology that incorporates automatic performance and primary metric data collection.

This section represents the evaluation of the methodological aspect of this general research objective and focuses on the extent to which the resultant PEEM achieves the objective of incorporating automatic performance and primary metric data collection. This issue is pivotal in establishing the practicality of this PEEM and the reproducibility of its evaluative results.

Phase 6 of the PEEM provides for the establishment of specifications for all evaluative criteria. The specification of Phase 7 of the PEEM establishes required primary metric data definitions, while Phase 4 provides for the identification and definition of any additional metrics. These environment-independent specifications provide the starting point for the development of the automatic performance and primary metric data collection facilities that are developed in Phase 9. While the specific implementation strategy for these facilities cannot be defined as part of this PEEM (since it is environment-specific), these phases have proved sufficient to guide the implementation of such facilities in a test case application of this PEEM (Section 5.1.4).

5.1.3 Evaluation of the Associated Specific Developmental Objective

To develop a prototype evaluation environment capable of supporting the proposed methodology.

As part of this research, a prototype environment for conducting evaluations under this PEEM was developed. As stated in Chapter 4, this environment was based on seven identical networked AT&T 7300 UNIX PCs with mice that were completely dedicated to these activities by the USL NASA Project. These systems supported the complete UNIX System V operating system, including all development utilities. Interface libraries for graphics and window control were available for the C language. Language support for both C and C++ was provided. The only capabilities not already established in these environments for support of the PEEM were automatic performance monitoring and primary metric recording and computation.

To establish the automatic performance monitoring capability on these workstations, several approaches to interactive transaction logging were prototyped, including a C-based transaction monitor program, a signal-based monitor based on multiple processes, and a shell-based monitor. Each of these monitors was inserted between the user and the operating system's default shell. Testing revealed that the shell-based approach imposed the least overhead since the operating system interpretation mechanism was always resident. Both the C-based approach and the signal-based approach caused excessive paging since they started new processes and competed with the operating system and user applications for residency. The shell-based approach was adopted based on this evaluation. Unique file name generation techniques were also developed to support the logging of collected data.

To establish the automatic primary metric data collection capability in this environment, LEX grammars were developed to scan both C and C++ source code. C support routines were also developed to augment the LEX capabilities with respect to token type determination.

With the establishment of these capabilities, the prototype environment was determined to be capable of supporting the PEEM. These activities served as the foundation for the evaluative execution (evaluated in Section 5.1.4) of the PEEM.

5.1.4 Evaluation of the Associated Specific Evaluative Objective

To conduct a systematic comparison of selected software development tasks in a specific applications domain using an object-oriented programming system with the same development tasks using a traditional programming system under the proposed methodology to demonstrate its evaluation strategy and the capabilities of this approach.

Chapter 4 of this document presented the test case application of the PEEM in the evaluation of development in C versus C++ for a highly interactive graphical application. This section focuses on the evaluation of this test case PEEM execution.

The test case PEEM execution presented in Chapter 4 was conducted over the course of two months utilizing seven graduate students in Computer Science. In particular, these students were involved in the development of the automatic monitoring facilities, the development of the automatic code analysis facilities, and the development of the base graphics editor applications. Four of these students were able to participate as subjects in the actual experiments. These substantial development activities represent a large part of the cost of executing the PEEM. The successful execution of the PEEM in this short amount of time, using only seven graduate students (including subjects) on a part-time basis, is indicative of the systematic and practical nature of this PEEM.

The automatic monitoring of development transactions and the automatic analysis of developed code served to reduce the amount of time and personnel needed to support the PEEM execution. If these facilities had been performed manually, this execution of the PEEM would have required at least four additional staff members for monitoring and classifying development transactions and another six staff members to assist in the manual analysis of the development transcripts and generated code. Needless to say, the results of such a manual process would be far less reliable and timely than those produced by this test case PEEM execution.

The PEEM execution produced extensive documentation of the evaluative context, the actual development processes, and the characteristics of the developed software (see Tables 4.1-4.7 and Appendices B-E). It is interesting to note that, while the scale of this evaluation (only four subjects) reflects the demonstrative nature of this execution of the PEEM, a larger scale evaluation could re-use all of the PEEM phase results, thereby significantly reducing the cost of PEEM re-execution.

The test case execution of the presented PEEM has successfully demonstrated its systematic and practical evaluation strategy. The capabilities and hence the potential value of this PEEM are also evidenced in the scope and detail of the information produced as a result of its execution (see Tables 4.1-4.7).

The only aspect of this general research objective not addressed in the evaluation of the associated specific research objectives is extensibility. Extensibility refers to the ability of the methodology to support extensions to the scope of its evaluative capabilities. Users of the methodology should be able to add new metrics, evaluate new features, and compare new environments. Existing programming environment evaluation methodologies were criticized in Section 3.2 for a lack of extensibility of this type.

The proposed methodology was defined in environment-generic terms and, accordingly, does not incorporate any specific environmental characteristics in establishing the evaluative framework. Specific provision has been made for identifying and incorporating additional metrics within the evaluation procedure, while comparability of results is preserved by requiring only a minimal set of primary metrics. These primary metrics are fundamentally based on the defining characteristics of object-oriented systems and, accordingly, are applicable to any newly-developed object-oriented system, regardless of additional features which may be included in such systems. The independence of the developed methodology of any specific experimental design allows even more fundamental extensibility.

This type of extensibility cannot be provided by evaluation methodologies that are based on environment-specific evaluation procedures, criteria and metrics, as are existing programming environment evaluation methodologies (see Section 3.2). The extensibility of the developed methodology is one of its strongest features.

The final consideration in the evaluation of this general research objective is this PEEM's support for long-term software complexity research for object-oriented systems. The degree to which the proposed methodology supports software complexity research for object-oriented systems depends on how completely it characterizes the aspects of software due to object-oriented design, on how consistently it can be applied, and on how well it can address emergent questions about object-oriented systems.

Specific design aspects of this methodology address these issues explicitly. The definition of primary software metrics is provided for the express purpose of characterizing unique object-oriented aspects of software. The systematic and experimental aspects of the methodology support consistent and repeatable evaluative results and comprehensive reliable data collection. Finally, the extensibility of the methodology permits the incorporation of developing aspects of software complexity research. We believe that the methodology developed as part of this research has met its initial goal of providing features supportive of metrics research for object-oriented systems; however, complete validation of the methodology must, in the final analysis, be based on extensive application in actual research and development settings.

5.2 Evaluation of the Application-Specific Paradigm General Research Objective

This section presents the evaluation of the application-specific paradigm general research objective identified in Section 2.2.

5.2.1 Evaluation of the Associated Specific Theoretical Objective

To determine fundamental design characteristics for specific applications domains that promote the theoretical validity of systematic comparisons of object-oriented systems.

This research has focused on the area of highly interactive graphical information systems in dealing with all application-domain specific issues. This focus was selected based on the high degree of object-oriented development activity involving these types of applications and to the particular long-standing interest of the USL NASA Project in this area. Consequently, the applications-specific object-oriented design characteristics developed in Section 4.3.3 focus on this specific area.

The fundamental object-oriented design characteristics identified in Section 4.3.2 were that truly object-oriented software must exhibit methods which are factored throughout the inheritance lattice and that each individual object must focus on a specific function or capability. In Section 4.3.3, these characteristics were refined to apply to interactive graphical applications, in particular. These application-specific characteristics included a distinct object associated with each graphical entity, methods and data structures factored throughout the application inheritance lattice, and objects partitioned between generic application-related and user interface-related functionality.

Since these applications-specific characteristics preserve the more general object-oriented software characteristics established in Section 4.3.2, software satisfying the applications-specific characteristics must be representative of object-oriented software technology. This prevents the pathological consideration of "one-object" software as being object-oriented and, thereby, promotes valid and systematic comparisons of software developed using this technology.

5.2.2 Evaluation of the Associated Specific Methodological Objective

To design applications domain-specific paradigms that support the effective application of object-oriented design techniques.

Applications domain-specific paradigms for object-oriented systems are guidelines for problem decomposition and software construction that preserve the fundamental object-oriented software characteristics of Section 4.3.2. In Section 4.3.3, application domain-specific characteristics and an associated paradigm were developed for highly interactive graphical applications. It was determined that only one paradigm was necessary to support the activities related to this research.

The individual guidelines of the paradigm mirror the desired characteristics directly but are phrased in terms of requirements for problem decomposition and software construction and include the requirement of separately instantiated objects with appropriate data structures and methods for each graphical entity, the requirement of sufficient private data structures to support all methods local to an object, the requirement that all common methods and data structures be factored into classes in the application inheritance lattice, and the requirement for a disjoint partitioning of generic application-related and user interface-related objects. The factoring, partitioning, and private data structures required in this paradigm force all inter-object communication to occur through messaging. The methods factored into the inheritance lattice are, by definition, re-used in the places from which they were factored. As shown in Section 4.3.3, if these requirements are satisfied, the fundamental object-oriented software characteristics of Section 4.3.2 will be preserved for highly interactive graphical applications.

5.2.3 Evaluation of the Associated Specific Developmental Objective

To develop procedures for the application of the applications domain-specific paradigms within a specific object-oriented development environment.

As stated in Section 4.2, the specific object-oriented environment selected for this research was AT&T's C++. This environment provides a full set of object-oriented facilities including a single-inheritance inheritance mechanism, a late-binding messaging facility (via virtual functions), and full encapsulation (via default *private* member type definitions). Since the application domain-specific paradigm of Section 4.3.3 was defined in terms of these basic object-oriented facilities, no translation into language specific procedures was necessary. For a language with limited object-oriented facilities (Ada, for example), some of the requirements of the application domain-specific paradigm may require the simulation of missing facilities, thereby requiring language-specific procedures. This may also be the case if the application domain-specific paradigm requires an intermediate model of execution (e.g., interprocess signaling, monitors, etc.) that is to be implemented with language-specific features.

In this research, the application domain-specific paradigm was directly applied to the test development situations. No difficulty was encountered in this approach in that the application specific paradigm directly mapped into the object-oriented facilities available within the selected environment.

5.2.4 Evaluation of the Associated Specific Evaluative Objective

To analytically verify that the applications-specific development paradigm does indeed constrain the resultant software products so that they are indeed representative of object-oriented designs.

The fundamental characteristics of object-oriented software, as identified in Section 4.3.2, are the degree to which methods are factored into the inheritance lattice and the degree to which an object focuses on a specific function or capability. Table 4.5 summarizes the object-oriented characteristics of the software generated using the application domain-specific paradigm of Section 4.3.3 pursuant to this research.

Observations from this table will suffice to establish that the fundamental object-oriented characteristics were preserved in the resultant software. Firstly, subjects 1, 2 and 4 added three classes each in the course of development. This included one class (object type) for each of the two geometric primitives and one class for the fill attribute capability. Subject 3 added a specialized menu class for the fill attribute primitive in addition to the other three classes and thus added a total of four classes. This observation directly demonstrates the second characteristic, namely, that objects focus on a specific function or capability. In this case, each added class provided exactly one function corresponding to one of the geometric primitives or to the fill attribute.

Secondly, to violate the factoring requirement, subjects would have had to implement only two classes, with each class replicating the features it needed. This would have resulted in one class that implemented filled and solid triangles and one class that implemented filled and solid rectangles; the fill attribute would not have been factored but redundantly implemented. As indicated above, this was not the case.

The evaluation of this specific research objective has demonstrated that the fundamental object-oriented software characteristics identified in Section 4.3.2 were preserved in software developed under the application domain-specific paradigm developed in Section 4.3.3 as part of this research.

In the evaluation of these specific research objectives, we have established that the application domain-specific paradigm (Section 4.3.3) does preserve the fundamental object-oriented software characteristics (Section 4.3.2) and we have established this to be the case for the specific software developed under this paradigm. The remainder of this section evaluates the effect of the application domain-specific paradigm on the design space available to the software developer.

Any large object can clearly be represented as a collection of smaller, more generic objects by constructing objects around data structures and transforming all references to those data structures into messages to the appropriate objects. Similarly, any inheritance lattice with redundantly-defined methods can be transformed into a factored lattice by establishing the appropriate class, and replacing the redundant methods by references to that class. Incidently, completely factored inheritance lattices can be transformed into unfactored inheritance lattices by substituting redundant implementations for factored class references, and small generic objects can be arbitrarily aggregated into larger more specialized objects, so that these transformations are bi-directional. Hence, the application domain-specific paradigm, in preserving object-oriented software characteristics, cannot restrict the space of possible designs available to the software developer. Any software developed under any other paradigm can be transformed to exhibit fundamental object-oriented characteristics.

5.3 Evaluation of the Primary Metrics General Research Objective

This section presents the evaluation of the primary metrics general research objective identified in Section 2.3.

5.3.1 Evaluation of the Associated Specific Theoretical Objective

To design primary metric data definitions that theoretically characterize the various aspects of software unique to object-oriented designs, including the inheritance lattice, the messaging graph, the degree of polymorphism exhibited, and degree of object re-use.

The primary metric data definitions developed as part of this research are identified in Section 3.4.7. They consist of the definition of two graphs that formally and completely establish the structural characteristics unique to object-oriented software (Section 3.4.7). The first of these graphs is the messaging graph, which represents the interaction between objects in a software system. The objects in a software system form the nodes of this graph and there exists a directed edge between two objects from the source object to the destination object for all messages exchanged between these nodes. This completely and formally defines the messaging graph in terms of object-oriented features. The second graph is the inheritance lattice, the navigation of which determines which specific methods and data structures comprise each class. The classes in a software system form the nodes in this graph and edges exist from any descendant class to its immediate parent class. This completely and formally defines the inheritance lattice in terms of object-oriented features.

The remaining object-oriented aspects of software can be defined in terms the aspects defined above. The degree of polymorphism is the number of methods defined for a particular object message. This can be defined since we know the method structure from the inheritance lattice. The degree of object re-use is the number of descendants which inherit the capabilities of that object. This can be directly determined from the inheritance lattice. Since these last two object-oriented software aspects can be determined from the inheritance lattice, only the inheritance lattice and the messaging graph need be captured during an evaluation.

5.3.2 Evaluation of the Associated Specific Methodological Objective

To develop language-independent methods for capturing this data for object-oriented designs. Language independence is demonstrated by constructing language-specific metric evaluation procedures for a representative set of object-oriented languages.

The generic procedures developed as part of this research for capturing both the messaging graph and the inheritance lattice were established in Section 3.4.7. The generic procedure for capturing the inheritance lattice is to determine, for every application class, its ancestors, descendants, and the defining class of any methods referenced within the class definition. For different languages, the syntactic structure of class definitions will be different, however the basic structure is the same. The sole function of any class definition is to establish any ancestors from which capabilities will be inherited and to permit the refinement or extension of those inherited capabilities. The generic procedure for capturing the messaging graph is to capture the source and destination of every message sent between objects within an application. As stated in

Section 3.4.7, the appropriate strategy for capturing this information is different for different languages and may require access to internal language mechanisms.

As demonstrated in Section 5.3.1, both degree of polymorphism and degree of re-use can be determined from the inheritance lattice. The degree of polymorphism can be determined by counting the number of different method definitions of a method within a class definition. Since all methods referenced within a class definition are captured as part of the inheritance lattice (Section 3.4.7), this information is immediately available. The degree of re-use for a class is determined by simply counting the number of classes that reference that class as an ancestor. This information is also immediately available as part of the inheritance lattice.

The evaluation of the language-independent aspect of these procedures is based on their ability to be instantiated as language-specific procedures for a representative set of object-oriented languages. The languages selected for this evaluation are CommonLoops, Objective-C, and Smalltalk. These languages are representative in that they are the most extensively used object-oriented languages available [Cox 1986] and they exhibit all of the current variations in object-oriented system features.

CommonLoops [Bobrow 1986] is a Common Lisp based object-oriented system and is becoming widely used in AI applications in Lisp-based workstations. In CommonLoops, the inheritance syntax is as follows:

(defstruct (new_class (:include (ancestor1, ancestor2,...))))

where *defstruct* signals a new class definition, *new_class* is the name of the newly defined class, *:include* is the inheritance function, and *ancestor1* and *ancestor2* are the names of the classes from which to inherit capabilities. The method definition syntax:

(defmethod new_class ((type arg1) arg1 arg2 ...) <code for method>)

establishes a new method for *new_class* that will be invoked when an object of type *new_class* gets a message of type *arg1*. Finally, the messaging syntax of CommonLoops is:

(send a 'object b)

where a message of type *a* is sent to *object* with arguments *a* and *b*.

The language specific procedure for determining the inheritance lattice for CommonLoops is as follows: recognize all *defstructs*, parse out the class being defined and any ancestors established, and record any methods defined for this class via *defmethod*. This procedure completely determines the inheritance lattice for CommonLoops, including any possible multiple inheritance relationships.

The messaging graph can be determined by modifying the *send* function to record the name of the calling function and the result of the standard CommonLoops expression

(methods-specified-by 'dest-object (type-of arg1))

(which returns the destination method) for each message.

Objective-C [Cox 1986] is an object-oriented extension to the C language developed by Productivity Products International and is widely used in user interface development. In Objective-C, the inheritance syntax is:

= NewClass : Ancestor { <additional data elements>;}

where *NewClass* is the name of the new class being created, *Ancestor* is the name of the parent class from which capabilities are inherited, and *<additional data elements>* refers to data elements added to those inherited. Additional methods are established as follows:

- NewMethod : Selector {<code for method>}

where *NewMethod* is associated with the last class defined previous to its own definition. A message of type *Selector* to an object of this class type will invoke this method. The messaging syntax for Objective-C is:

[Obj Selector:arg1]

which sends a message of type *Selector* to the object *Obj* with the argument *arg1.*

The language-specific procedure for determining the inheritance lattice for Objective-C is as follows: recognize all class definitions (this is trivial since only class definition statements begin with =), parse out the name of the class being defined and the ancestor established, and record any methods defined for this class (also trivial since only method definition statements begin with -). This procedure completely determines the inheritance lattice for Objective-C.

The messaging graph can be determined by modifying the internal Objective-C function *_msg* to record the name of the sending object and the destination method resolved, for all messages.

Xerox's Smalltalk [Goldberg 1983] was one of the first object-oriented programming languages and set the standard for integrated iconic user interfaces.

Smalltalk environments provide different textual representations of the inheritance construct, since inheritance is normally dealt with through the interactive browser facilities and not in a textual form. The syntax used here was defined by Timothy Bud of Oregon State University [Bud 1987]. Any other textual syntax for Smalltalk inheritance would be very similar. The Smalltalk inheritance syntax is:

```
Class NewClass : Ancestor | <additional data elements> |
    [
    <method definitions>
    ]
```

where *NewClass* is the name of the new class being created, *Ancestor* is the name of the parent class from which capabilities are inherited, and *<additional data elements>* refers to data elements (known as instance variables) added to those inherited. *Method definitions* are established as follows:

NewMethod : Selector | <temporary variables> | <code for method>|

A message of type *Selector* to an object of class *NewClass* will invoke this method. The messaging syntax for Smalltalk is:

Obj Selector:arg1

which sends a message of type *Selector* to the object *Obj* with the argument *arg1.*

The language specific procedure for determining the inheritance lattice for Smalltalk, then, is as follows: recognize all class definitions via the *Class* keyword, parse out the name of class being defined and the ancestor established, and record any methods defined for this class which are identified by the a colon following the method name within the method definition section. This procedure completely determines the inheritance lattice for Smalltalk.

All statements within Smalltalk methods are message expressions. The messaging graph can be determined by inserting additional message expressions for each of these lines which send the *class* message to *Obj* and to the current object via the *self* reference and recording the returned results for all messages.

These language-specific procedures demonstrate the instantiation of the PEEM's language-independent primary metric data capture procedures for a representative set of object-oriented languages.

5.3.3 Evaluation of the Associated Specific Developmental Objective

To provide language-specific acquisition of this metric data for the object-oriented development systems under consideration in the evaluation environment.

As part of the activities of Phase 9 (Section 4.9) of the test case execution of the presented PEEM, the automatic facilities for capturing the primary metric data were implemented for C++. The automatic facilities for capturing the traditional metrics for C++ were also designed for use on C code. Both sets of facilities were based on LEX grammars; however, the memory requirements of the resulting code was very large. This may indicate a problem with this approach for more complex languages. The LEX grammars are included in Appendices D and E.

5.3.4 Evaluation of the Associated Specific Evaluative Objective

To comprehensively test these metric data acquisition methods for accuracy.

The testing of these primary metric computation facilities was conducted on the National Institute of Health's OOPS (object-oriented programming support) library developed by Keith Gorlen [Gorlen 1987]. This library consists of over 2500 lines of commercially developed C++ code and implements the C++ equivalent of all of the non-graphical Smalltalk classes. This library represents extensive coverage of available C++ language features. The facilities for C were tested on the post-processed form of this same library.

The results of these tests were verified manually for randomly selected C++ classes and no runtime errors were detected in the final version of the primary metric computation facilities, indicating more than adequate sizes for static internal data structures generated by LEX.

The evaluation of these specific research objectives has established the design, development, application, and validation of the these primary metric data definitions and facilities. These facilities have played a pivotal role in establishing both the capability and the practical usability of the presented PEEM. The advantages of reliability and consistency in these facilities have far outweighed the initial investment in their development.

CHAPTER 6: SUMMARY AND CONCLUSIONS

This concluding chapter summarizes the research presented in previous chapters, identifies the significant contributions contained therein, establishes the major conclusions of this work, and establishes research directions that are direct extensions of this work.

6.1 Summary

This section will follow the organization of this dissertation, summarizing each chapter individually.

Chapter 1 established an identification of the problem that has motivated this research, namely, the insufficiency of available research data and facilities to support complexity model and metric research for object-oriented systems. This chapter also provided an overview of object-oriented activities in a broad set of applications domains as empirical substantiation of the productivity potential of object-oriented systems technology. Chapter 1 concluded with an identification of the three general objectives of this research (Section 1.3).

Chapter 2 provided a refinement of each of the general research objectives into specific theoretical, methodological, developmental, and evaluative objectives. This chapter also established the significance of the each of the general research objectives and the attainment criteria for each of the supportive specific research objectives.

Chapter 3 overviewed the existing approaches to programming environment evaluation and identified the specific weaknesses of these approaches with respect to supporting the consistent evaluation of object-oriented systems.

This chapter continued with the identification of the methodology design principles intended to address these weaknesses. Chapter 3 concluded with a complete specification of the individual phases of the developed PEEM, including an identification of related design principles and specific phase execution guidelines.

Chapter 4 presented a demonstrative test case application of the methodology which consisted of a comparative evaluation of highly interactive graphical software development in C and C++. This test case was intended only as a concrete demonstration of the capabilities of this methodology. The results of the evaluation, despite the small scale of the experiment, were consistent with the observations made in Chapter 1 concerning the positive impact of object-oriented techniques on development productivity.

Chapter 5 presented a detailed evaluation of the degree of attainment of each of the general and specific research objectives. Chapter 5 also established that this research has indeed fulfilled both the general research objectives identified in Section 1.3 and the specific research objectives identified in Chapter 2.

6.2 Summary of Research Contributions

This section summarizes the major contributions of the research presented in this document. These contributions closely reflect certain of the research objectives identified in Sections 2.1, 2.2, and 2.3. These major contributions are as follows:

1. This research has formally established the primary metric data definitions that completely characterize the unique aspects object-oriented software

systems, including the inheritance lattice and messaging graph.

2. This research has established language-independent procedures for automatically capturing this primary metric data during an evaluation. These procedures have been shown to be instantiable in a representative set of object-oriented languages.

3. This research has established the fundamental characteristics of object-oriented software that indicate consistent applications of object-oriented design techniques, namely, that common capabilities are factored throughout the inheritance lattice and that individual objects focus on providing specific capabilities.

4. This research has defined a language-independent application domain-specific development paradigm based on these fundamental characteristics for highly interactive graphical applications.

5. This research has identified design principles for a programming environment evaluation methodology that ensure its applicability to object-oriented development environments. The PEEM design principles unique to this work include the following: the requirement for primary metric data definitions that completely characterize the object-oriented characteristics of the software under evaluation, the requirement for the identification of relevant applications domain-specific development paradigms to support the validity and comparability of evaluative results, and the requirement for automatic capture of performance and primary metric data to ensure consistency and eliminate human bias.

6. Finally, this research has produced a systematic, extensible, and environment-independent programming environment evaluation methodology capable of supporting research into complexity models and metrics for object-oriented systems. The design principles, identified in contribution 5 above, establish the basis of the fundamental distinctions between exiting PEEMs and the PEEM developed as part of this research.

6.3 Major Conclusions

This research has developed a programming environment evaluation methodology which is unique in its ability to support consistent and repeatable evaluations of the productivity implications of object-oriented software development environments and which provides a strategic mechanism for supporting research into complexity models and metrics for software development in such environments. The methodology incorporates two unique design concepts which are pivotal in supporting these characteristics, namely, the applications domain-specific development paradigms to support consistent and valid comparative evaluations and the primary metric data definitions to support the complete characterization of the object-oriented aspects of developed software. The systematic, extensible and environment-independent nature of the presented methodology has been established by analysis and demonstrated by test case execution of the methodology.

6.4 Identification of Future Research Directions

As stated in Chapter 1, the primary motivation of this research was, and continues to be, the support of long-term research into complexity models and metrics appropriate to object-oriented systems. We feel that the research presented within this document represents a very significant step toward this strategic goal and provides a solid mechanism for pursuing emerging research issues with respect to object-oriented systems.

The most direct extension of the work presented here is the large scale application of this methodology to additional evaluation contexts. This would, of course, include evaluations within additional applications domains, the incorporation of alternate applications domain-specific development paradigms, and the extension of evaluations to include other object-oriented development environments.

The comparative investigation of the productivity impact of various extensions to, and refinements of, the object-oriented paradigm would, likewise, be a natural extension of this work. Issues that could be addressed based on currently proposed object-oriented system extensions include the investigation of the productivity impact of:

1. Alternate inheritance mechanisms, including various forms of strict inheritance.

2. Inheritance lattice organizational strategies (e.g., name space partitioning).

3. Messaging graph organizational strategies (e.g., message protocols).

4. Object and message protocol specification and consistency mechanisms (e.g., method post- and pre-conditions, class consistency constraints; see [Meyer 1987]).

5. Parallel computation extensions to the object-oriented paradigm (e.g.,

Actors; see [Agha 1986]).

6. Object-oriented distributed processing mechanisms (e.g., Orient84/K; see [Ishikawa 1986]).

Finally, the most significant direct extension of this work involves the formulation of cognitive complexity models of software development using object-oriented techniques, the development of appropriate software metrics for these models, and the subsequent validation of these metrics and the associated models via application of this methodology. Proposed models must attempt to account for the currently inadequately understood phenomena associated with development in object-oriented environments, including the very significant productivity improvements in a wide variety of application domains and the very long developer learning curves for large scale object-oriented environments (e.g., the Symbolics' Flavors system). Models which successfully account for these phenomena will certainly improve our ability to apply object-oriented technology and may provide a mechanism for significantly advancing our understanding of the fundamentals of the software development process as a whole.

BIBLIOGRAPHY

[Agha 1986] Agha, Gul A. *ACTORS: A Model of Concurrent Computation in Distributed Systems,* Cambridge Press, Cambridge, Massachusetts, 1986.

[Anderson 1986] Anderson, David B. "Experience with Flamingo: A Distributed, Object-Oriented User Interface System," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 177-184.

[Bailey 1985] Bailey and Kramer, J.F. *A Framework for Evaluating the Usability of Programming Support Environments,* Computer and Software Engineering Division, Institute for Defense Analysis, IDA Paper TR P-1942, November, 1985.

[Balzer 1986] Balzer, R. and Goldman, N. "Principles of Good Software Specification and their Implications for Specification Languages," *Software Specification Techniques,* Addison-Wesley, Reading, Massachusetts, 1986, pp. 25-39.

[Basili 1981a] Basili, V. and Phillips, T. "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Performance Evaluation Review,* Vol. 10, March 1981, pp. 95-106.

[Basili 1981b] Basili, V. and Reiter, R.W. "A Controlled Experiment Quantitatively Comparing Software Development Approaches," *IEEE Transactions on Software Engineering,* May 1981, pp. 299-320.

[Basili 1986] Basili, V., Selby, W.R. and Hutchens, D.H. "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering,* Vol. 12, July 1986, pp. 733-743.

[Birtwistle 1984] Birtwistle, G.M. "Future Directions in Simulation Software," Panel Discussion. In, *Proceedings of an SCS Conference,* La Jolla, 1984, Bryant, R. and Ungers, B.W. (eds.), pp. 120-121.

[Black 1986] Black, A., Hutchinson, N., Jul, E. and Levy, H. "Object Structure in the Emerald System," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 78-86.

[Bobrow 1986] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F. "CommonLoops: Merging Lisp and Object-Oriented Programming," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 17-29.

[Boehm 1981] Boehm, B.W. *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Brinker 1985] Brinker, E. *An Evaluation of the Softech, Inc. Ada Language System, Version 1.5.,* NASA/GSFC Code 522.1, NASA, December 1985.

[Bud 1987] Bud, T. *A Little Smalltalk,* Addison-Wesley, Reading, Massachusetts, 1985.

[Castor 1983] Castor, V.L. *Criteria for the Evaluation of ROLM Corporation's Ada Work Center,* Air Force Wright Aeronautical Laboratories, January 1983.

[Chikayama 1984] Chikayama, T. *ESP Reference Manual,* Technical Report TR 044, ICOT, 1984.

[Conte 1986] Conte, S.D., Dunsmore, H.E. and Shen V.Y. *Software Engineering Metrics and Models,* Benjamin Cummings, Menlo Park, California, 1986.

[Cox 1986] Cox, B.J. *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley, Boston, Massachusetts, 1986.

[Dahl 1966] Dahl, O.J. and Nygaard, K. "SIMULA - An ALGOL-based Simulation Language," *CACM,* Vol. 9, No. 9, pp. 671-678.

[Dasgupta 1986] Dasgupta, Partha. "A Probe-Based Monitoring Scheme for an Object-Oriented, Distributed Operating System," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 57-66.

[Dominick 1987] Dominick, W.D. "Annotated Index of USL NASA/RECON and NASA/JPL Project Publications," Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, La., Updated: April 22, 1987, 33p. (This document represents an annotated index of the 80+ publications and reports that have resulted from NASA-funded projects at USL from December, 1983 to the present.)

[Ewing 1986] Ewing, Juanita J. "An Object-Oriented Operating System Interface," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 46-56.

[Fikes 1985] Fikes, R. and Kehler, T. "The Role of Frame-based Representation in Reasoning," *CACM,* Vol. 28, No. 9, pp. 904-920.

[Franta 1973] Franta, W.R. and Maly, K. *The Suitability of a Very High Level Language (SETL) for Simulation and Control,* Technical Report TR 73-4, Department of Computer Science, University of Minnesota, 1973.

[Fukunaga 1986] Fukunaga, Koichi and Shin-ichi Hirose. "An Experience with a Prolog-Based Object-Oriented Language," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 224-231.

[Gannon 1977] Gannon, J.D. "An Experimental Evaluation of Data Type Conventions," *CACM,* Vol. 20, No. 8, August 1977, pp. 584-595.

[Goldberg 1983] Goldberg, A. and Robson, D. *SMALLTALK-80: The Language and Its Implementation,* Addison-Wesley, Boston, Massachusetts, 1983.

[Gorlen 1986] Gorlen, K. *Object-Oriented Program Support: Reference Manual,* National Institutes of Health, Bethesda, Maryland, 1986.

[Halstead 1977] Halstead, M.H. *Elements of Software Science,* Elsevier North-Holland, New York, 1977.

[Harrison 1985] Harrison, W. "A Method of Sharing Industrial Software Complexity Data," *ACM: Sigplan Notices,* February 1985, pp. 42-51.

[Helsgaun 1980] Helsgaun, K. "DISCO - A SIMULA-based Language for Continuous, Combined and Discrete Simulation," *SIMULATION,* July 1980, pp. 1-12.

[Hewitt 1973] Hewitt, C., Bishop, P. and Steiger, R. "A Universal Modular ACTOR Formalism for Artificial Intelligence," *Proceedings on the 3rd International Joint Conference on Artificial Intelligence,* August, 1973.

[Hook 1985] Hook, A.A., Riccardi, G.A., Vilot, M. and Welke, S. *User's Manual for Prototype Ada Compiler Evaluation Capability, Version 1,* Institute for Defense Analysis, IDA Paper TR P-1879, October, 1985.

[Ishikawa 1986] Ishikawa, Yutaka and Tokoro, Mario. "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementations," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 232-241.

[JonesC 1986] Jones, Capers. *Programming Productivity,* McGraw-Hill Inc., New York, 1986.

[JonesM 1986] Jones, Michael B. and Rashid, Richard F. "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 67-77.

[Kaehler 1985] Kaehler, Ted and Patterson, Dave. "A Small Taste of Smalltalk," *BYTE: The Small Systems Journal,* August 1985, pp. 145-159.

[Kahn 1986] Kahn, Kenneth, Tribble, E.D., Miller, M.S., and Bobrow, D.G. "Objects in Concurrent Logic Programming Languages," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 242-257.

[Kreutzer 1986] Kreutzer, W. *System Simulation: Programming Styles and Languages,* Addison-Wesley, Boston, Massachusetts, 1986.

[Lang 1986] Lang, Kevin J. and Pearlmutter, Barak A. "OAKLISP: An Object-Oriented Scheme with First Class Types," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 30-37.

[Lindquist 1985] Lindquist, T.E. "Assessing the Usability of Human-Computer Interfaces," *IEEE Software,* Vol. 2, No. 1, January 1985, pp. 74-82.

[Maier 1986] Maier, David, Stein, J., Otis, A. and Purdy, A. "Development of an Object Oriented DBMS," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 472-482.

[Maier 1985] Maier, David, Otis, A. and Purdy, A. "Object-Oriented Database Development at Servio Logic," *Database Engineering Bulletin,* Vol. 8, No. 4, December 1985, pp. 58-65.

[McCabe 1976] McCabe, T.J. "A Complexity Measure," *IEEE Transactions on Software Engineering,* Vol. 2, No. 4, December, 1976, pp. 308-320.

[Meyer 1987] Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design," *IEEE Software,* Vol. 4, No. 2, March 1987, pp. 50-64.

[Meyer 1986] Meyer, Bertrand. "Genericity Versus Inheritance," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 391-405.

[Meyrowitz 1986] Meyrowitz, Norman. "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 186-201.

[Moon 1986] Moon, David A. "Object Oriented Programming With Flavors," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 1-8.

[Nakashima 1984] Nakashima, H. "Knowledge Representation in Prolog/KR," *Proceedings of the 1984 International Symposium on Logic Programming,* 1984, pp. 126-130.

[Novak 1980] Novak, G.S. "Data Abstraction in GLISP," *Proceedings of the ACM Conference on Principles of Programming Languages,* ACM, 1980, pp. 170-177.

[Nygaard 1986] Nygaard, Kristen. "Basic Concepts in Object Oriented Programming," *ACM: Sigplan Notices,* October 1986, pp. 128-132.

[Papazoglou 1984] Papazoglou, M.R. "An Outline of the Programming Language SIMULA," *Computer Languages,* Vol. 9, No. 2, pp. 107-131.

[Pascoe 1985] Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," *BYTE: The Small Systems Journal,* August 1985, pp. 139-144.

[Roberts 1983] Roberts, T.L. and Moran, T.P. "The Evaluation of Text Editors: Methodology and Empirical Results," *CACM,* Vol. 26, No. 49, pp. 265-283.

[Schmucker 1985] Schmucker, Kurt J. "Object-Oriented Languages for the Macintosh," *BYTE: The Small Systems Journal,* August 1985, pp. 177-185.

[Sim 1975] Sim, K. *CADSIM Users' Guide and Reference Manual,* Imperial College Publishers, London, England, 1975.

[Skarra 1986] Skarra, Andrea H. and Zdonik, Stanley B. "The Management of Changing Types in an Object-Oriented Database," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 483-495.

[Shneiderman 1987] Shneiderman, B. *Designing the User Interface,* Addison-Wesley, Boston, Massachusetts, 1987.

[Soloway 1984] Soloway, E. and Ehrlich, K. "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering,* Vol. 10, September 1984, pp. 595-609.

[Stefik 1986] Stefik, M. and Bobrow, D.G. "Object-Oriented Programming: Themes and Variations," *AI Magazine,* Vol. 6, No. 4, 1986, pp. 40-62.

[Stroustrup 1986] Stroustrup, B. *The C++ Programming Language,* Addison-Wesley, Boston, Massachusetts, 1986.

[Tesler 1985] Tesler, Larry. "Programming Experiences," *BYTE: The Small Systems Journal,* August 1985, pp. 195-206.

[Wegner 1986] Wegner, Peter. "Classification in Object-Oriented Systems," *ACM: Sigplan Notices,* October 1986, pp. 173-182.

[Weiderman 1987] Weiderman, N.H., Habermann, A.N., Borger, M.W., and Klein, M.H. "A Methodology for Evaluating Environments," *ACM: Sigplan Notices,* January 1987, pp. 199-207.

[Weinreb 1981] Weinreb, D. and Moon, D. *LISP Machine LISP Manual* (4th edition), MIT Press, Cambridge, Massachusetts, 1981.

[Weissman 1974] Weissman, L. "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM: Sigplan Notices,* June 1974, pp. 25-36.

[Yokote 1986] Yokote, Yasuhiko and Tokoro, Mario. "The Design and Implementation of ConcurrentSmalltalk," *OOPSLA'86: Sigplan Notices Special Issue,* November 1986, pp. 331-340.

APPENDIX A

TRADEMARKS REFERENCED IN THIS DOCUMENT

Ada is a trademark of the U.S. Department of Defense

Amiga is a trademark of Commodore-Amiga, Inc.

Atari ST is a trademark of Atari Computers

ESP is a trademark of Expert Systems International.

FLAVORS is a trademark of Symbolics, Inc.

KEE is a trademark of Intellicorp.

Macintosh is a trademark of Apple Computer, Inc.

Microsoft is a trademark of Microsoft Corporation

Objective-C is a trademark of Productivity Products International.

SMALLTALK is a trademark of XEROX Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

APPENDIX B

PROFICIENCY TESTS AND ANSWER KEYS

This appendix documents the C++ and VDI proficiency tests that were administered during the experiment execution phase, Phase 10 Section 4.10, of the developed programming environment evaluation methodology. The structure of this document is as follows: Section B1 contains the C++ proficiency test, Section B2 contains the C++ answer key, Section B3 contains the VDI proficiency test, and Section B4 contains the VDI answer key. All section references within the C++ and VDI answer key sections are from Stroustrup's *The C++ Programming Language* published by Addison-Wesley; see [Stroustrup 1986] in the bibliography section of this dissertation document.

B1. C++ Proficiency Test

Instructions: Circle the most appropriate choice. Only one choice may be selected for each question.

Each question references a given program.
Each program uses the same include file, "classdef" which accompanies the test.

1) Select the most correct choice.

```
#include "classdef"
            main()
            {
      (1)   apple b;
      (2)   banana y(3);
      (3)   orange o;
            }
```

a) Statement (1) cannot be invoked.
b) Statement (2) cannot be invoked.
c) Statement (3) cannot be invoked.
d) Statements (1), (2), and (3) cannot be invoked.
e) Statements (1), (2), and (3) CAN be invoked.

2) Select the most correct choice.

a) The "classdef" program fragment,
"apple operator+(apple& a) {return (apple(num() + a.num()));}"
produces the same results as
"apple operator+(apple& a) {return (apple(number + a.number));}".

b) The "classdef" program fragment,
"banana operator+(banana& a) {return(banana(a.num() + num()));}"
produces the same results as
"banana operator+(banana& a) {return (banana(number + a.number));}".

c) The "classdef" program fragment,
"orange operator+(orange& a){return(a.onum + onum);}"
produces the same results as
"orange operator+(orange& a) {return (orange(number + a.number));}".

d) Choices b) and c) are true.
e) Choices a) and b) are true.

3) Select the most correct choice.

```
    #include "classdef"
            main()
            {
    (1)     apple a(1);
    (2)     banana b;
    (3)     orange o;
    (4)     a.print();
    (5)     b.print();
    (6)     o.print();
            }
```

a) Statement (4) produces the output "Number of fruit: 1".
b) Statement (5) produces the output "Number of banana: 0".
c) Statement (6) produces the output "Number of fruit: 0".
d) Choices a) and c) are true.
e) Choices a) and b) are true.

4) Select the value of the "ap2" number field in statement (6).

```
    #include "classdef"
            main()
            {
    (1)     apple ap1(1);
    (2)     apple ap2(2);
    (3)     apple ap3(3);
    (4)     ap2 = 4;
    (5)     ap1 = ap3;
    (6)     ap2 = ap1 + ap3;
            }
```

a) 3.
b) 6.
c) 4.
d) 5.
e) Has no value because one or more statements cannot be done.

5) Select the value of the "b2" number field in statement (3).

```
#include "classdef"
        main()
        {
(1)     banana ba;
(2)     banana ban = 4;
(3)     banana b2(ba+ban);
        }
```

a) 2.
b) 4.
c) 0.
d) 3.
e) Has no value because one or more statements cannot be done.

6) Select the value of the "ba" number field in statement (4).

```
#include "classdef"
        main()
        {
(1)     banana ba;
(2)     banana b2(8);
(3)     banana ban = 4;
(4)     ba = b2 + 2 + ban;
        }
```

a) 2.
b) 10.
c) 4.
d) 14.
e) Has no value because one or more statements cannot be done.

7) Select the value of the "or3" number field in statement (5).

```
#include "classdef"
        main()
        {
(1)     orange  or, or1, or2;
(2)     or = 3;
(3)     or2 = 4;
(4)     or1 = 2;
(5)     orange or3 = or + or1 * or2;

        }
```

a) 20.
b) 9.
c) 11.
d) 0.
e) Has no value because one or more statements cannot be done.

8) Select the results of statement (4).

```
#include "classdef"
        main()
        {
(1)     apple a(5);
(2)     apple& pp = a;
(3)     pp = 1;
(4)     a.print();
        }
```

a) Number of fruit: 1.
b) Number of fruit: 6.
c) Number of fruit: 7.
d) Number of fruit: 5.
e) Has no value because one or more statements cannot be done.

9) Select the most correct choice.

```
#include "classdef"
        main()
        {
(1)     fruit *xx;
(2)     fruit f;
(3)     xx = &f;
(4)     banana *yy;
(5)     banana b;
(6)     yy = &b;
(7)     apple *zz;
(8)     apple a(0);
(9)     zz = &a;
(10)    (*xx).print();
(11)    yy->print();
(12)    *zz.print();
        }
```

a) Statement (10) compiles and produces the same results as "f.print()".
b) Statement (11) compiles and produces the same results as "b.print()".
c) Statement (12) compiles and produces the same results as "a.print()".
d) Choices a) and b) are true.
e) Choices a) and c) are true.

10) Select the most correct choice.

```
#include "classdef"
        main()
        {
(1)     apple a(2);
(2)     banana b;
(3)     orange o;
(4)     if (a.num() < 0) {};
(5)     if (b.num() < 0) {};
(6)     if (o.num() < 0) {};
        }
```

a) Statement (4) is a legal statement.
b) Statement (5) is a legal statement.
c) Statement (6) is a legal statement.
d) Statement (4) and (5) are legal statements.
e) Statement (4), (5), and (6) are ALL ILLEGAL statements.

11) Select the value of the "b2" number field in statement (4).

```
#include "classdef"
        main()
        {
(1)     banana ba;
(2)     banana b2(13);
(3)     banana ban = 2;
(4)     b2 = 7 + ba + ban;
        }
```

a) 22.
b) 0.
c) 17.
d) 9.
e) Has no value because one or more statements cannot be done.

12) Select the value of the "o" number field in statement (4).

```
#include "classdef"
        main()
        {
(1)     orange o;
(2)     orange o1(22);
(3)     orange o2(11);
(3)     apple  a(8);
(4)     o = o1 + a + o2;

        }
```

a) 30.
b) 11.
c) 41.
d) 8.
e) Has no value because one or more statements cannot be done.

----------------** End of test **---------------

Class Definitions for this Test

```
#include <stream.h>

class fruit {
                friend orange;
                int number;
        public:
                friend banana;
                fruit() {number = 0;}
                fruit(fruit& k) {number = k.number;}
                fruit(int i) {number = i;}
                int num() {return number;}
                int operator=(int j) {return number = j;}
                virtual void print() {cout << "Number of fruit: " << number << "0;}
};

class apple : public fruit {
                int num1;
        public:
                apple(int i) : (i) {};
                apple(apple& k) : (k) {};
                apple operator+(apple& a) {return (apple(num() + a.num()));}
                int operator=(int j) { return fruit :: operator=(j);}
 };

class banana : fruit {
        public:
                banana () {};
                banana(int i) : (i) {};
                banana operator+(banana& a){return(banana(a.num() + num()));}
                int operator=(int j) { return fruit :: operator=(j);}
                virtual void print() {cout << "Number of banana: " << number << 0;}
};

class orange {
                int onum;
        public:
                orange() {onum = 0;}
                orange(int i) {onum = i;}
                orange(apple& a) {onum = a.num();}
                int operator=(int j) { return onum = j ;}
                orange operator+(orange& a){return(a.onum + onum);}
                orange operator*(orange& a){return(a.onum * onum);}


};
```

B2. C++ Answer Key

****
Question One
****

Major Testing Focus	:	Rules of inheritance between a base class and other classes with respect to the public members. The other classes are: public derived, derived friend, and friend.
Answers	:	(a) Recognizes the rules of inheritance between a base class and other classes. Recognizes the concept of constructors.
		(b) Fails to recognize that a derived class does not inherit the constructor of its base class. Fails to recognize valid and invalid constructors.
		(c) Fails to recognize that a derived class does not inherit the constructor of its base class. Fails to recognize valid constructors.
		(d) Fails to recognize valid constructors.
		(e) Fails to recognize that a derived class does not inherit the constructor of its base class.
Correct Answer	:	a
References	:	All section references are to Stroustrup's *The C++ Programming Language* [Stroustrup 1986].

1.13 Derived Classes (pg 30)
5.2.2 Classes (pg 136)
6.10 Friends and Members (pg 187)
7.2 Derived Classes (pg 192)
8.5.5 Constructors (pg 278)
8.5.9 Visibility of Member Names (pg 281)
8.5.10 Friends (pg 281)

****
Question Two
****

Major Testing Focus	:	Rules of inheritance between a base class and other classes with respect to the private members. The other classes are: public derived, derived friend, and friend.
Answers	:	(a) Fails to recognize that a public derived type cannot access the private part of its base class. Fails to recognize the rules of inheritance with regard to friend classes.
		(b) Recognizes the rules of inheritance between a base class and other classes.
		(c) Fails to recognize that a friend declaration can be placed in either the private or the public part of a class declaration and/or fails to recognize that a friend class can only use the private variables that are defined for that class.
		(d) Fails to recognize that a friend class can only use the private variables that are defined for that class.
		(e) Fails to recognize that a derived class cannot access the private variable(s) of a base.
Correct Answer	:	b
References	:	1.13 Derived Classes (pg 30)
		5.2.2 Classes (pg 136)
		6.10 Friends and Members (pg 187)
		7.2 Derived Classes (pg 192)
		8.5.5 Constructors (pg 278)
		8.5.9 Visibility of Member Names (pg 281)
		8.5.10 Friends (pg 281)

****
Question Three
****

Major Testing Focus	:	Rules of virtual functions between a base class and other classes. The other classes are: public derived, derived friend, and friend.	
Answers	:	(a)	Fails to recognize that a virtual function can be redefined in a derived class.
		(b)	Fails to recognize the concept of virtual functions. Only acknowledged the "print" member function.
		(c)	Fails to recognize the concept of virtual functions.
		(d)	Fails to recognize the concept of virtual functions.
		(e)	Recognizes that only a derived class can use the base class' virtual function when the derived class has not defined its own version.
Correct Answer	:	e	
References	:	1.18	Virtual Functions (pg 37)
		7.2.8	Virtual Functions (pg 201)
		8.5.4	Virtual Functions (pg 277)
		8.5.5	Constructors (pg 278)

****
Question Four
****

Major Testing Focus	:	User-defined Type Conversion	
Answers	:	(a)	Logical arithmetic error.
		(b)	Recognizes the user-defined addition operator and the bitwise copying of objects.
		(c)	Logical arithmetic error.
		(d)	Logical arithmetic error.
		(e)	Fails to recognize that statement (4) can be accomplished through a bitwise copying of objects.
Correct Answer	:	b	
References	:	1.14	More about Operators (pg 32)
		1.8	Operator Overloading (pg 25)
		6.0	Operator Overloading (pg 169)
		6.3	User-defined Type Conversion (pg 173)
		6.6	Assignment and Initialization (pg 178)

****
Question Five
****

Major Testing Focus :	Messaging and Operator Overloading	
Answers :	(a)	Logical arithmetic error.
	(b)	Recognizes that a bitwise copy is implicit since a banana(banana&) constructor doesn't exist.
	(c)	Logical arithmetic error.
	(d)	Logical arithmetic error.
	(e)	Fails to recognize that a banana(banana&) is not required because an implicit bitwise copy is done.
Correct Answer :	b	
References :	1.14	More about Operators (pg 32)
	1.8	Operator Overloading (pg 25)
	2.3.10	References (pg 56)
	5.5	Constructors and Destructors (pg 157)
	6.3	User-defined Type Conversion (pg 173)

****
Question Six
****

Major Testing Focus	:	Messaging and Operator Overloading	
Answers	:	(a)	Logical arithmetic error.
		(b)	Logical arithmetic error.
		(c)	Logical arithmetic error.
		(d)	Recognizes that the compiler can construct a banana object from "2" only because the left to right evaluation identified the operation as a banana operation.
		(e)	Fails to recognize that the banana can construct an object from "2" and/or fails to recognize that a user-defined addition operation for a banana and integer addition was not necessary. Fails to recognize the a left to right evaluation.
Correct Answer	:	d	
References	:	1.14	More about Operators (pg 32)
		1.8	Operator Overloading (pg 25)
		3.2	Operator Summary (pg 84)

****
Question Seven
****

Major Testing Focus	:	Messaging and Operator Precedence	
Answers	:	(a)	Fails to recognize operator precedence.
		(b)	Logical arithmetic error.
		(c)	Recognized operator precedence and user defined types.
		(d)	Logical arithmetic error.
		(e)	Fails to recognize that "operator+" receives a "temporary" orange object containing the results of "operator*".
Correct Answer	:	c	
References	:	1.14	More about Operators (pg 32)
		1.8	Operator Overloading (pg 25)
		3.2	Operator Summary (pg 84)

****
Question Eight
****

Major Testing Focus	:	References	
Answers	:	(a)	Recognized the concept of references.
		(b)	Fails to recognize an assignment over an addition to "a".
		(c)	Fails to recognize a reference pointer, constructor, and assignment operator.
		(d)	Fails to recognize assignment of "1" to "pp" as an assignment to "a".
		(e)	Fails to recognize the declaration of a reference pointer.
Correct Answer	:	a	
References	:	2.3.10 References (pg 56)	

****

Question Nine

****

Major Testing Focus	:	Pointers	
Answers	:	(a)	Fails to recognize equivalent and non-equivalent pointer expressions to virtual functions.
		(b)	Fails to recognize equivalent and non-equivalent pointer expressions to virtual functions.
		(c)	Fails to recognize equivalent and non-equivalent pointer expressions to virtual functions.
		(d)	Recognizes equivalent and non-equivalent pointer expressions to virtual functions.
		(e)	Fails to recognize equivalent and non-equivalent pointer expressions to virtual functions.
Correct Answer	:	d	
References	:	1.18	Virtual Functions (pg 37)
		7.2.4	Pointers (pg 197)
		7.2.8	Virtual Functions (pg 201)
		8.5.4	Virtual Functions (pg 277)
		8.5.5	Constructors (pg 278)

****
Question Ten
****

Major Testing Focus	:	Scoping and Self Reference	
Answers	:	(a)	Recognizes that a public derived class can access the public members of its base class.
		(b)	Fails to recognize that a derived friend class can access public members of its base class only with its body and not externally.
		(c)	Fails to recognize that a friend class can only access the public members of the "base" class within its body and not externally.
		(d)	Fails to recognize that a derived class can only use the public members of its base externally when the derived class declares a public base class.
		(e)	Fails to recognize the access of member functions outside of the object's body.
Correct Answer	:	a	
References	:	5.2.3	Self Reference (pg 137)

****
Question Eleven
****

Major Testing Focus	:	Messaging and Operator Overloading	
		Operator Precedence	
Answers	:	(a)	Logical arithmetic error.
		(b)	Logical arithmetic error.
		(c)	Logical arithmetic error.
		(d)	Fails to recognized that a banana object could not be constructed because an integer and banana addition operator had not been defined. The left to right evaluation would not allow the operation to be defined within the banana class.
		(e)	Recognizes that the left to right evaluation would not allow the operation to be defined within the banana object.
Correct Answer	:	e	
References	:	1.14	More about Operators (pg 32)
		1.8	Operator Overloading (pg 25)
		3.2	Operator Summary (pg 84)

****
Question Twelve
****

Major Testing Focus	:	User-defined Type Conversions	
Answers	:	(a)	Logical arithmetic error.
		(b)	Logical arithmetic error.
		(c)	Recognizes that the orange class has a constructor that can convert an apple into an orange so that the "operator+" can add an apple and an orange.
		(d)	Logical arithmetic error.
		(e)	Fails to recognize the conversion of the apple to an orange so that the "operator+" can add an apple and an orange.
Correct Answer	:	c	
References	:	1.14	More about Operators (pg 32)
		1.8	Operator Overloading (pg 25)
		3.2	Operator Summary (pg 84)
		8.5.5	Constructors (pg 278)

B3. VDI Proficiency Test

Instructions: Circle the most appropriate choice. Only one choice may be selected for each question.

Some questions reference the C and C++ graphics editors which will be provided with appropriate documentation. Other questions will reference the VDI and TAM functions and the respective reference manuals will be available.

1) Which VDI command should be used in order to clear the workstation screen ?

a) winit();

b) wcreate();

c) system("clear");

d) v_clrwk();

e) The commands are not VDI and do not clear the screen.

2) A device handle is:

a) A number that uniquely identifies a specific device so that one or more devices may be opened simultaneously.

b) Initialized by v_opnwk().

c) Used in subsequent VDI calls to identify a specific device.

d) Choices a), b), and c) are true.

e) Choices a), and c) are true.

3) When adding a new line style (e.g. dash twodots) to the line style menu in the C version of the graphics editor, the following modules must be modified:

   a) kern.c

   b) attr.c

   c) main.c

   d) Choices a) and c) are true.

   e) Choices a) and b) are true.

4) How many unique line types are supported on the AT&T 7300?

   a) 2 -> Green and Black.

   b) 7 -> Solid, Long dashed, Dotted, Dashed-dotted, Medium dashed, Dashed with Two Dots, and Short Dash.

   c) N -> User defined within the limits of the ASCII Table.

   d) 4 -> Hollow, Solid, Hatch, and Pattern.

   e) 6 -> Solid, Long dashed, Dotted, Dashed-dotted, Medium dashed, and Dashed with Two Dots.

5) The difference(s) between GET_MOUSE_POINT and GET_MOUSE_POINT_LINE functions in the C graphics editor is/are:

a) GET_MOUSE_POINT_LINE accepts a point and a button and returns a new point. GET_MOUSE_POINT accepts a button and returns a point.

b) GET_MOUSE_POINT_LINE draws a line and GET_MOUSE_POINT draws a point.

c) GET_MOUSE_POINT_LINE accepts a button and returns two points. GET_MOUSE_POINT accepts a button and returns one point.

d) Choices a) and b).

e) Choices b) and c).

6) In the C graphics editor, which of the functions below should be used to get the initial point of a polygon?

a) GET_MOUSE_POINT_LINE()

b) GET_MOUSE_POINT()

c) GET_MOUSE_POINT_BOX()

d) Choice b) followed by choice a)

e) Choices a), b), c), and d) are false.

7) Which C graphics editor function would be equivalent to the C++ graphics editor function, "get_mouse(point p)" ?

   a) GET_MOUSE_POINT_LINE()

   b) GET_MOUSE_POINT()

   c) GET_MOUSE_POINT_BOX()

   d) Choices a), b), and c) are true.

   e) Choices a), b), c), and d) are false.

8) After a polyline is drawn in the C and C++ graphics editor, the vertices of the polyline:

   a) Do not exist any more in the C graphics editor.

   b) Remain as an instantiation of polyline_object in the C++ graphics editor.

   c) Cannot be referenced in C++ graphics editor.

   d) Choices a), b), and c) are true.

   e) Choices a), b), and c) are false.

9) The design of the C++ program provides an interface to the VDI commands in:

a) kern_ic.c

b) gfxed.c

c) obj_ic.c

d) menu_ic.c

e) attr_ic.c

****
The next three questions will test your understanding of the program organization of the ***C++ GRAPHICS EDITOR***. The questions will revolve around the insertion of a "hexagon" primitive and should be answered within the design constraints of the program.
****

10) In order to have the "hexagon" choice appear in the main menu, which module must be modified:

a) gfxed.c

b) menu_ic.c

c) menu_ic.h

d) kern_ic.c

e) Choices b) and c) are true.

11) The "hexagon" primitive requires a class declaration. Where should that declaration be placed:

a) kern_ic.c

b) obj_ic.h

c) obj_ic.c

d) Choices b) and c).

e) Choices a), b), and c).

12) The "hexagon" class member functions that are defined outside of the class declaration should be placed in:

a) kern_ic.c

b) obj_ic.h

c) obj_ic.c

d) Choices b) and c).

e) Choices a), b), and c).

B4. VDI Answer Key

****
Question One
****

Major Testing Focus	:	Distinguish VDI primitive, TAM primitive, and System Call	
Answers	:	(a)	Fails to recognize the difference between a TAM and VDI primitive. Fails to recognize that the TAM primitive does not clear the screen.
		(b)	Fails to recognize the difference between a TAM and VDI primitive. Recognizes that the TAM primitive clears the screen.
		(c)	Fails to recognize the difference between a System call and VDI primitive. Recognizes that the System call clears the screen.
		(d)	Recognized the VDI primitive and its functionality.
		(e)	Fails to recognize a VDI primitive and/or its functionality.
Correct Answer	:	d	
References	:	AT&T VDI Programmer's Guide 5-7 AT&T User's Manual Volume II TAM(3T) C Kernighan and Ritchie 157	

****
Question Two
****

Major Testing Focus :	Understanding of the device handle.	
Answers :	(a)	Recognizes that a device handle can simultaneously reference more than one device because of the unique value. Fails to recognize where the value is initialized and that it is required for subsequent VDI calls.
	(b)	Recognizes that the device handle receives the unique value from the invocation of opnwk(). Fails to recognize that the device handle can reference multiple devices and its subsequent use in VDI calls.
	(c)	Recognizes that the device handle references a specific device in VDI calls. Fails to recognize that the device handle is used to differentiate between open devices. Fails to recognize that its value is obtained from opnwk().
	(d)	Recognizes the concept of device handle.
	(e)	Fails to recognize that v_opnwk() initializes the variable, device handle.
Correct Answer :	d	
References :	AT&T VDI Programmer's Guide 2-16	
	AT&T VDI Programmer's Guide G-3	

****
Question Three
****

Major Testing Focus	:	Design of the C Graphics Editor with respect to the task of adding a line style	
Answers	:	(a)	Fails to recognize that the kern.c module is for VDI primitives
		(b)	Recognizes that the line style menu (line style type) is defined within the attr.c module
		(c)	Fails to recognize that the menu for line style is contained within attr.c
		(d)	Fails to recognize that line style is a drawing attribute.
		(e)	Fails to recognize that attr.c defines the domain of attributes that kern.c may use in subsequent calls. Recognizes that a data structure in attr.c contains the available line attributes.
Correct Answer	:	b	
References	:	AT&T VDI Programmer's Guide 5-87 AT&T VDI Programmer's Guide C-12 C Graphics Editor	

****
Question Four
****

Major Testing Focus	:	Design of the C Graphics Editor with respect to the definition of the VDI vsl_type().	
Answers	:	(a)	Fails to recognize the difference between a line type and color
		(b)	Recognizes that the line type is device dependent. For the AT&T, the number of types supported is seven.
		(c)	Fails to recognize that line type is device dependent.
		(d)	Fails to recognize the difference between a line type and interior fill style.
		(e)	Recognizes line types. However, fails to recognize that the line types are device dependent, and that the AT&T 7300 provides more than the standard 6 line types.
Correct Answer	:	b	
References	:	AT&T VDI Programmer's Guide 5-87	
		AT&T VDI Programmer's Guide C-12	

****
Question Five
****

Major Testing Focus :	Basic "Point" Functions of the C Graphics Editor	
Answers :	(a)	Recognizes that GET_MOUSE_POINT_LINE requires input parameters, point and button, and returns a new point. Recognizes that GET_MOUSE_POINT accepts a button and returns a point.
	(b)	Fails to recognize that the function does not draw a line but rather assists in the acquisition of a second point by rubberbanding. Fails to recognize that the function does not draw a point but rather returns a point.
	(c)	Fails to recognize that GET_MOUSE_POINT_LINE returns only one point. Recognizes that GET_MOUSE_POINT returns only one point.
	(d)	Fails to recognize the function of GET_MOUSE_POINT_LINE and GET_MOUSE_POINT.
	(e)	Fails to recognize the function of GET_MOUSE_POINT_LINE and GET_MOUSE_POINT.
Correct Answer :	a	
References :	AT&T VDI Programmer's Guide 5-104 C Graphics Editor	

****
Question Six
****

Major Testing Focus	:	Initial Construction of a Polygon or Arc
Answers	:	(a) Fails to recognize that an initial point is required for the function to operate or that GET_MOUSE_LINE expects a (prior) point to draw from.
		(b) Recognizes that the GET_MOUSE_POINT function is used to get an initial point.
		(c) Fails to recognize that an initial point is required for the function to operate.
		(d) Fails to recognize the function of GET_MOUSE_POINT_LINE and GET_MOUSE_POINT.
		(e) Fails to recognize the function of GET_MOUSE_POINT_LINE and GET_MOUSE_POINT.
Correct Answer	:	b
References	:	AT&T VDI Programmer's Guide 5-104 C Graphics Editor

****
Question Seven
****

Major Testing Focus	:	C equivalent of the C++ overloaded function, "get_mouse(point p)".	
Answers	:	(a)	Recognizes that when get_mouse is called with a point parameter, it rubberbands a line to identify the point to be returned.
		(b)	Fails to recognize that get_mouse is an overloaded function and when passed a point as a parameter refers to the GET_MOUSE_POINT_LINE.
		(c)	Fails to recognize that when get_mouse() is called with no parameters, it only returns a point and that it doesn't rubberband.
		(d)	Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(e)	Fails to understand the question.
Correct Answer	:	a	
References	:	C++ Graphics Editor	

****
Question Eight
****

Major Testing Focus	:	The difference in scope of the polyline object between the C and C++ graphics editors.	
Answers	:	(a)	Recognizes that the C version uses automatic vertex array in the DO_POLY routine and when DO_POLY is out of scope, the information is lost.
		(b)	Recognizes that when polylines are drawn, a polyline object is instantiated and that the object remains after the object is drawn.
		(c)	Recognizes that even though the object still exists, it cannot be referenced because the object's pointer value is not saved.
		(d)	Recognizes the concept of object instantiation and object scope.
		(e)	Fails to recognize the concept of object instantiation and object scope.
Correct Answer	:	d	
References	:	C++ Graphics Editor	

****
Question Nine
****

Major Testing Focus	:	Design of the C++ Graphics Editor with respect to the of VDI commands.
Answers	:	(a) Recognizes that the kern_ic.c is the interface for VDI commands.
		(b) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(c) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(d) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(e) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
Correct Answer	:	a
References	:	C++ Graphics Editor

****
Question Ten
****

Major Testing Focus :	Design of the C++ Graphics Editor with respect to the task of adding a "hexagon" choice to the main menu.	
Answers :	(a)	Recognizes that the modular design of the C++ graphics editor instantiates the main menu in gfxed.c
	(b)	Fails to recognize that the menu_ic.c contains the menu class member functions for menu operations.
	(c)	Fails to recognize that the menu_ic.h contains the menu class declaration and not the actual objects.
	(d)	Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
	(e)	Fails to recognize that the menu modules are only for class definition and member functions.
Correct Answer :	a	
References :	C++ Graphics Editor	

****
Question Eleven
****

Major Testing Focus	:	Design of the C++ Graphics Editor with respect to the task of adding a "hexagon" class definition.
Answer	:	(a) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(b) Recognizes that the modular design of the C++ graphics editor requires the class declaration of the "hexagon" within the obj_ic.h.
		(c) Fails to recognize that the obj_ic.c contains the member functions of the "hexagon" class and not the class declaration.
		(d) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(e) Fails to recognize the modular design of the C++ graphics editor.
Correct Answer	:	b
References	:	C++ Graphics Editor

****
Question Twelve
****

Major Testing Focus	:	Design of the C++ Graphics Editor with respect to the task of adding a "hexagon" class member functions.
Answer	:	(a) Fails to recognize that the kern_ic.c module provides a level of abstraction from the VDI primitives.
		(b) Fails to recognize that the obj_ic.h should only contain the class declaration of the "hexagon".
		(c) Recognizes that the modular design of the C++ graphics editor requires the member function of the "hexagon" within the obj_ic.c.
		(e) Fails to recognize the modular design of the C++ graphics editor.
Correct Answer	:	c
References	:	C++ Graphics Editor

APPENDIX C

PERFORMANCE MONITOR DOCUMENTATION

This appendix documents the performance monitor mechanism implemented to support the test case experiment execution phase (Phase 10 of Section 4.10) of the developed programming environment evaluation methodology. The structure of this appendix is as follows: Section C1 contains the definitions of all collected data associated with the performance monitor; Section C2 contains the source for the monitor script as implemented for the AT&T 7300 UNIX PC under System V; Section C3 contains a brief and partial excerpt from the resulting transaction log.

C1. Definitions of Collected Data

The following data elements are collected for each transaction with the system (all times are in hour:minute:second format):

Begin time	The time at which the return key was typed at the end of the current command line.
Command	The text of the current command as typed by the subject. This command will subsequently be passed on to a shell for execution.
User Name	The user identification associated with the subject for the current command. This identification is a coded reference to the subject. This element is tracked on a per transaction basis to enable the detection of changes in user identity between transactions (e.g., super user, root, etc.).
Finish Time	The time at which the command, dispatched to the execution shell, has completed and control is returned to the user.

The following set of data elements are used as documentation of the transaction logging session itself and include the time, date, and user name that were in effect when the monitor was invoked.

File Name	This is a unique file name generated by concatenating the string *log.* with the numeric values of the month, day, hour, and minute at which the monitor was invoked. This is the filename under which the current monitored data is stored.
File Owner	This data item is maintained by the operating system for the transaction log file and is accessible to all analysis programs. It serves as an external identifier of the User Name associated with the current subject and is always the User Name at monitor invocation. The subject cannot change this value as he can the User Name above. This permits the identification of a subject's transaction logs from a directory listing.
Creation Time	This data item documents the time at which the monitor was invoked. Subjects were permitted to read the task specification and commence at this time (this was signaled by a system prompt).

C2. MONITOR SCRIPT SOURCE

```
trap "" 2
DATE=‘date '+%m%d%H%M'‘
LOGFILE="/u/exp/log.$DATE"
while :
do
        echo "unixpc%          read command argument
        case $command in
                "logout") break
                ;;
                "") ;;
                "cd")
                        echo "0 >> $LOGFILE
                        echo "Begin time :‘date‘" >> $LOGFILE
                        echo "Command     : $command $argument" >> $LOGFILE
                        echo "User Name  : $USERNAME" >> $LOGFILE
                        $command $argument;
                        echo "Finish time: ‘date‘" >> $LOGFILE
                        ;;
                *)
                        echo "0 >> $LOGFILE
                        echo "Begin time : ‘date‘" >> $LOGFILE
                        echo "Command     : $command $argument" >> $LOGFILE
                        echo "User Name  : $USERNAME" >> $LOGFILE
                        ksh -c "$command $argument";
                        echo "Finish time: ‘date‘" >> $LOGFILE
                        ;;
                esac
done
kill -9 $$
```

C3. TRANSACTION LOG EXCERPT

Data Maintained by the Operating System

```
Filename          :     log.07231647
File Owner        :     subject2
Creation Time     :     16:47:30
```

Data Logged by Monitor

```
Begin time        :     16:47:35
Command           :     cd gfxed_c
User Name         :     subject2
Finish time       :     16:47:35

Begin time        :     16:47:37
Command           :     ls
User Name         :     subject2
Finish time       :     16:47:39

Begin time        :     16:47:47
Command           :     vi obj.c
User Name         :     subject2
Finish time       :     16:53:45

Begin time        :     16:53:47
Command           :     make
User Name         :     subject2
Finish time       :     16:55:38

Begin time        :     16:55:41
Command           :     gfed
User Name         :     subject2
Finish time       :     16:56:10
```

APPENDIX D

LEX GRAMMAR FOR OBJECT ORIENTED PRIMARY METRIC DATA ANALYSIS

This appendix documents the object-oriented primary metric data capture mechanism implemented to support the test case execution of Phase 9 (Section 4.9) of the developed programming environment evaluation methodology. The relevant primary metric data definitions are presented in Section 3.4.7. This appendix contains the LEX grammar developed to support determination of the inheritance lattice. As stated in Section 4.9, the messaging graph mechanism was not implemented due to vendor delays in delivery of the C++ translator source.

LEX Grammar for C++ Analysis

```
%{
#include "y.tab.h"
extern int class, class_head, braces, fin_fun_param, start_garbage;
extern int class_member_defs, num_classes, total_num_members;
extern int total_lines_per_class, lines_per_class, on, off;
%}
D     [0-9]
I     [a-zA-Z]
H     [0-9a-fA-F]
V     [0-9a-zA-Z_]
O     [0-7]
P     [1-9]
L     [lL]
U     [uU]
E     [eE]
ESC   [abfnrtv'"?\\]
X     [xX]
A   "#"
C   "//"
F   "\n"
M   "/*"
N   "*/"
%%
"#"[ \n]*"\n"    {}
"//"[ \n]*"\n"   {}
{M}[ {N}]*{N}    {}
"asm"        { return(check(ASM));}
"auto"       { return(check(AUTO));}
"break"      { return(check(BREAK));}
"case"       { return(check(CASE));}
"char"       { return(check(CHAR));}
"cin" { return(check(IDENTIFIER));}
"class"      { return(check(CLASS));}
"continue"   { return(check(CONTINUE));}
"cout"       { return(check(IDENTIFIER));}
"default"    { return(check(DEFAULT));}
"delete"     { return(check(DELETE));}
"do" { return(check(DO));}
"double"     { return(check(DOUBLE));}
"else"       { return(check(ELSE));}
"enum"       { return(check(ENUM));}
"extern"     { return(check(EXTERN));}
"float"      { return(check(FLOAT));}
"for" { return(check(FOR));}
"friend"     { return(check(FRIEND));}
"goto"       { return(check(GOTO));}
"if"  { return(check(IF));}
"inline"     { return(check(INLINE));}
"int" { return(check(INT));}
"long"       { return(check(LONG));}
"new"        { return(check(NEW));}
"operator"   { return(check(OPERATOR));}
"overload"   { return(check(OVERLOAD));}
"public"     { return(check(PUBLIC));}
"register"   { return(check(REGISTER));}
"return"     { return(check(RETURN));}
"short"      { return(check(SHORT));}
"signed"     { return(check(SIGNED));}
"sizeof"     { return(check(SIZEOF));}
"static"     { return(check(STATIC));}
```

```
"struct"    { return(check(STRUCT));}
"switch"    { return(check(SWITCH));}
"this"      { return(check(THIS));}
"typedef"   { return(check(TYPEDEF));}
"union"     { return(check(UNION));}
"unsigned"  { return(check(UNSIGNED));}
"virtual"   { return(check(VIRTUAL));}
"void"      { return(check(VOID));}
"volatile"  { return(check(VOLATILE));}
"while"     { return(check(WHILE));}
{I}{V}*     { return(check(IDENTIFIER));}
0{X}{H}+{L}?{U}?   { return(check(CONSTANT));}
0{X}{H}+{U}?{L}?   { return(check(CONSTANT));}
0{O}+{L}?{U}? { return(check(CONSTANT));}
0{O}+{U}?{L}? { return(check(CONSTANT));}
{P}{D}*{L}?{U}?      { return(check(CONSTANT));}
{P}{D}*{U}?{L}?      { return(check(CONSTANT));}
\'([ '\n\\]|(\\({ESC}|{X}{H}{H}?{H}?|{O}{O}?{O}?)))+\'
       { return(check(CONSTANT));}
\"([ "\n\\]|(\\({ESC}|{X}{H}{H}?{H}?|{O}{O}?{O}?)))+\"
       { return(check(STRING_LITERAL));}
{D}+{E}{L}?    { return(check(CONSTANT));}
{D}*"."{D}+{E}?{L}?        { return(check(CONSTANT));}
{D}+"."{D}*{E}?{L}?        { return(check(CONSTANT));}
"..."  { return(check(ELLIPSIS));}
">>="    { return(check(RIGHT_ASSIGN));}
"<<="    { return(check(LEFT_ASSIGN));}
"+="     { return(check(ADD_ASSIGN));}
"-=" { return(check(SUB_ASSIGN));}
"*="     { return(check(MUL_ASSIGN));}
"/="     { return(check(DIV_ASSIGN));}
"%="     { return(check(MOD_ASSIGN));}
"&="     { return(check(AND_ASSIGN));}
" =" { return(check(XOR_ASSIGN));}
"|=" { return(check(OR_ASSIGN));}
">>"     { return(check(RIGHT_OP));}
"<<"     { return(check(LEFT_OP));}
"++"     { return(check(INC_OP));}
"--"  { return(check(DEC_OP));}
"->" { return(check(PTR_OP));}
"&&"     { return(check(AND_OP));}
"||"  { return(check(OR_OP));}
"<="     { return(check(LE_OP));}
">="     { return(check(GE_OP));}
"=="     { return(check(EQ_OP));}
"!=" { return(check(NE_OP));}
";"   { return(check(SMCLN));}
"{"   { return(check(CRBRO));}
"}"   { return(check(CRBRC));}
","   { return(check(COMMA));}
":"   { return(check(FLCLN));}
"="   { return(check(EQUAL));}
"("   { return(check(PARNO));}
")"   { return(check(PARNC));}
"["   { return(check(SQBRO));}
"]"   { return(check(SQBRC));}
"."   { return(check(PRIOD));}
"&"   { return(check(AMPSD));}
"!"   { return(check(EXCLM));}
"~ "  { return(check(NEGAT));}
"-"   { return(check(MINUS));}
"+"   { return(check(PLUSS));}
"*"   { return(check(MULT));}
"%"   { return(check(PRSNT));}
"<"   { return(check(LSTHN));}
```

```
">"  { return(check(GRTHN));}
" "   { return(check(CARET));}
"|"   { return(check(ORSYM));}
"?"   { return(check(QUSTN));}
[ \t\v\n\f] { count_lines();}
%%
yywrap()
/*----- SYSTEM ROUTINE ------*/
{
return(1); }

int count_lines()
/* ------------------------------------------------------------+
 COUNT LINES PER CLASS.
+--------------------------------------------------------------*/
{
   int i;
   extern int trace;
   if  (class)
      if (yytext[0]=='\n')
          lines_per_class++;
}
int check(value)
/* ------------------------------------------------------------+
CHECK THE CURRENT VALUE AND PERFORM APPROPRIATE ACTIONS.
+--------------------------------------------------------------*/
int value;
{
  if  (trace) printf("\nclass(%d)token(%s)\n",class,yytext);

  if  ( (!class)&&(value==CLASS) )
        { class=on;
          class_head=on;
          num_classes++;
          class_member_defs=0;
          lines_per_class=1;
          return(CLASS);
        }
                                        if (trace) printf("1 \n");
  if  (!class)
      return(GARBAGE);
                                        if (trace) printf("2 \n");

  if  (class_head)
      { if  (value==CRBRO)
            {class_head=off;
             braces++;
            }
        return(value);
      }
                                        if (trace) printf("3 \n");
   if  (value==CRBRO)
        {++braces;
         return(value);
        }
                                        if (trace) printf("4 \n");
   if  (value==CRBRC)
        if  (--braces <= 1)
            return (CRBRC);

                                        if (trace) printf("5 \n");
   if  (braces > 1)
        return(STUFF);

                                        if (trace) printf("6 \n");
```

```
    if  ((value==SMCLN)&&(braces==0))
        { fin_fun_param = class = class_head = off;
          total_num_members = total_num_members + class_member_defs;
          total_lines_per_class = total_lines_per_class + lines_per_class;
          return (SMCLN);
        }
                                        if (trace) printf("7 \n");
    if (value==PARNO)
      {start_garbage=on;
       class_member_defs++;
       return(value);
      }
                                        if (trace) printf("8 \n");
    if (value==PARNC)
        {fin_fun_param=on;
         start_garbage=off;
         return(value);
        }
                                        if (trace) printf("9 \n");
    if ((fin_fun_param)&&(value==SMCLN))
       {fin_fun_param=off;
        return(value);
       }
                                        if (trace) printf("10 \n");
    if (value==SMCLN)
        return(SMCLN);
                                        if (trace) printf("11 \n");
    if (start_garbage)
        return(PARAMS);
                                        if (trace) printf("12 \n");
    if ((value==FLCLN)||(value==PUBLIC)||(value==FRIEND)
                                    ||(value==CLASS))
          return(value);
                                        if (trace) printf("13 \n");
    return (STUFF);
}
```

Rudimentary Driver

```
%{
char class_name[15];
int xxx, yylineno;
int on=1 , off= 0;
int class=0, fin_fun_param=0, num_classes=0, start_garbage=0;
int class_member_defs=0, class_def=0, class_head=0, braces=0;
int lines_per_class=1, total_num_members=0, total_lines_per_class=0;
char buff[500];
%}
%start file
%union { char rest[5000];
       }
%token <rest>  AUTO
%token <rest>  BREAK
%token <rest>  CASE
%token <rest>  CHAR
%token <rest>  CLASS
%token <rest>  CONST
%token <rest>  CONTINUE
%token <rest>  DEFAULT
%token <rest>  DELETE
%token <rest>  DO
%token <rest>  DOUBLE
%token <rest>  ELSE
%token <rest>  ENUM
%token <rest>  EXTERN
%token <rest>  FLOAT
%token <rest>  FOR
%token <rest>  FRIEND
%token <rest>  GOTO
%token <rest>  IF
%token <rest>  INLINE
%token <rest>  INT
%token <rest>  LONG
%token <rest>  NEW
%token <rest>  OPERATOR
%token <rest>  OVERLOAD
%token <rest>  PUBLIC
%token <rest>  REGISTER
%token <rest>  RETURN
%token <rest>  SHORT
%token <rest>  SIGNED
%token <rest>  SIZEOF
%token <rest>  STATIC
%token <rest>  STRUCT
%token <rest>  SWITCH
%token <rest>  THIS
%token <rest>  TYPEDEF
%token <rest>  UNION
%token <rest>  UNSIGNED
%token <rest>  VIRTUAL
%token <rest>  VOID
%token <rest>  VOLATILE
%token <rest>  WHILE
%token <rest>  CONSTANT
%token <rest>  STRING_LITERAL
%token <rest>  ASM
%token <rest>  IDENTIFIER
%token <rest>  ELLIPSIS
%token <rest>  RIGHT_ASSIGN
%token <rest>  LEFT_ASSIGN
%token <rest>  ADD_ASSIGN
%token <rest>  SUB_ASSIGN
```

```
%token <rest>  MUL_ASSIGN
%token <rest>  DIV_ASSIGN
%token <rest>  MOD_ASSIGN
%token <rest>  AND_ASSIGN
%token <rest>  XOR_ASSIGN
%token <rest>  OR_ASSIGN
%token <rest>  RIGHT_OP
%token <rest>  LEFT_OP
%token <rest>  INC_OP
%token <rest>  DEC_OP
%token <rest>  PTR_OP
%token <rest>  AND_OP
%token <rest>  OR_OP
%token <rest>  LE_OP
%token <rest>  GE_OP
%token <rest>  EQ_OP
%token <rest>  NE_OP
%token <rest>  SMCLN
%token <rest>  CRBRO
%token <rest>  CRBRC
%token <rest>  COMMA
%token <rest>  FLCLN
%token <rest>  EQUAL
%token <rest>  PARNO
%token <rest>  PARNC
%token <rest>  SQBRO
%token <rest>  SQBRC
%token <rest>  PRIOD
%token <rest>  AMPSD
%token <rest>  EXCLM
%token <rest>  NEGAT
%token <rest>  MINUS
%token <rest>  PLUSS
%token <rest>  MULT
%token <rest>  DIVIS
%token <rest>  PRSNT
%token <rest>  LSTHN
%token <rest>  GRTHN
%token <rest>  CARET
%token <rest>  ORSYM
%token <rest>  QUSTN
%token <rest>  ENUM_CONST
%token <rest>  TYPEDEF_NAME
%token <rest>  GARBAGE
%token <rest>  FOR_LATER
%token <rest>  DEFOP
%token <rest>  STUFF
%token <rest>  PARAMS
%type <rest>  identifier
%type <rest>  file
%type <rest>   program
%type <rest>  garbage
%type <rest>  class_specifiers
%type <rest>  class_head
%type <rest>  class_specifier
%type <rest>  class_descr
%type <rest>  i_am_in_a_class
%type <rest>  what_we_need
%type <rest>  fun_body
%type <rest>  what_we_do_not_need
%type <rest>  some_stuff
%type <rest>  some_params
%type <rest>  decls
%type <rest>  decls2
%type <rest>  function_name
```

```
%type <rest>  type_and_identifier
%type <rest>  some_decl
%type <rest>  rest_of_function
%type <rest>  function_params1
%type <rest>  function_params2
%type <rest>  typedef_name for_later references
%type <rest>  operator defined_op sc_specifier
%%
file
        : program
          {printf("\n\t\t\t------------------------------\n");
           printf("\t\t\t    C++ METRICS ANALYZER\n");
           printf("\t\t\t FOR INHERITANCE AND MESSAGING\n");
           printf("\t\t\t------------------------------\n");
           printf("\t\t\t * TOTAL # CLASSES (%d)\n",num_classes);
           printf("\t\t\t * AVG. MEMBERS/CLASS (%2.2f)\n",
                  (float)total_num_members/(float)num_classes);
           printf("\t\t\t * AVG. CODED LINES/CLASS (%2.2f)\n",
                  (float)total_lines_per_class/(float)num_classes);
           printf("\n%s\n",$1,strlen($1));}
        ;
program
        : garbage  {strcpy($$,"");}
        | class_specifiers
          {sprintf(buff,"\n%s--# OF MEMBERS (%d)--# LINES (%d)--\n\n",
                 $1,class_member_defs,lines_per_class);
                        strcpy($$,buff);}
        | program class_specifiers
          {sprintf(buff,"\n%s%s--# OF MEMBERS (%d)--#LINES (%d)--\n\n",
                 $1,$2,class_member_defs,lines_per_class);
           strcpy($$,buff);}
        | program garbage {sprintf(buff,"%s",$1); strcpy($$,buff);}
        ;
garbage
        : GARBAGE {strcpy($$,"");}
        ;
class_specifiers
        : class_head CRBRO CRBRC  SMCLN
          {sprintf(buff,"%s ",$1);strcpy($$,buff);}
        | class_head CRBRO class_descr CRBRC SMCLN
          {sprintf(buff,"%s%s",$1,$3);strcpy($$,buff);}
        | class_head CRBRO class_descr PUBLIC FLCLN class_descr
             CRBRC SMCLN
          {sprintf(buff,"%s%sPUBLIC\n%s",
                        $1,$3,$6);strcpy($$,buff);}
        | class_head CRBRO PUBLIC FLCLN class_descr CRBRC SMCLN
          {sprintf(buff,"%sPUBLIC\n%s",$1,$5);strcpy($$,buff);}
        ;
class_descr
        : i_am_in_a_class {strcpy($$,$1);}
        ;
i_am_in_a_class
        : what_we_do_not_need{strcpy($$,"");}
        | what_we_need
          {sprintf(buff,"%s",$1); strcpy($$,buff);}
        | i_am_in_a_class what_we_need
          {sprintf(buff,"%s %s",$1,$2); strcpy($$,buff);}
        | i_am_in_a_class what_we_do_not_need
          {sprintf(buff,"%s",$1); strcpy($$,buff);}
        ;
what_we_do_not_need
        : some_stuff SMCLN
          {strcpy($$,"");}
        ;
what_we_need
```

```
        : FRIEND some_stuff SMCLN
          {sprintf(buff,"friend %s;\n",$2); strcpy($$,buff);}
        | CLASS some_stuff SMCLN
          {sprintf(buff,"class %s;\n",$2); strcpy($$,buff);}
        | some_stuff PARNO PARNC SMCLN
          {sprintf(buff,"%s ();\n",$1); strcpy($$,buff);}
        | some_stuff PARNO PARNC fun_body
          {sprintf(buff,"%s ();\n",$1); strcpy($$,buff);}
        | some_stuff PARNO some_params PARNC SMCLN
          {sprintf(buff,"%s(%s)\n",$1,$3); strcpy($$,buff);}
        | some_stuff PARNO some_params PARNC fun_body
          {sprintf(buff,"%s(%s)\n",$1,$3); strcpy($$,buff);}
        ;
fun_body   : CRBRO  CRBRC
        | CRBRO some_stuff CRBRC
        | CRBRO CRBRC SMCLN
        | CRBRO some_stuff CRBRC SMCLN
        ;
some_params
        : PARAMS
          {strcpy($$,yytext);}
        | some_params PARAMS
          {sprintf(buff,"%s%s",$1,yytext);strcpy($$,buff);}
        ;
some_stuff : STUFF
          {strcpy($$,yytext);}
        | some_stuff STUFF
          {sprintf(buff,"%s %s",$1,yytext); strcpy($$,buff);}
        ;
class_head
        : CLASS {strcpy($$,"CLASS:\nBASE:\n");}
        | CLASS identifier
          {sprintf(buff,"CLASS: %s\nBASE:\n",$2); strcpy($$,buff);}
        | CLASS identifier FLCLN identifier
            {sprintf(buff,"CLASS: %s\nBASE: %s\n",$2,$4);
             strcpy($$,buff); }
        | CLASS identifier FLCLN PUBLIC identifier
          {sprintf(buff,"CLASS: %s\nBASE: PUBLIC %s\n",$2,$5);
           strcpy($$,buff);}
        ;
```

APPENDIX E

LEX GRAMMAR FOR TRADITIONAL PRIMARY METRIC DATA ANALYSIS

This appendix documents the traditional primary metric data capture mechanism implemented to support the test case execution of Phase 9 (Section 4.9) of the developed programming environment evaluation methodology. The relevant primary metric data definitions are presented in Section 3.4.7. This appendix contains the LEX grammar developed to support determination of these traditional metrics.

LEX Grammar for Traditional Metric Analysis

```
%{
#include "y.tab.h"
#include "lex.ext.h"
#include <stdio.h>
#include <strings.h>
#include <math.h>
#define SL 200
#define ID 40
#define OT 200
#define OD 400
#define NC 100
main()
{
    int i,index,cindex,tokval,noc,cci;
    double N,V,eta;
    struct pac {
        int numtokoccur;
        char tokenid[SL];
    };
    struct   {
        char cname[ID];
        int  N1,N2;
        int  numofprdcts;
        int  numofoprnds;
        int  numofoprtrs;
        struct pac  operators[OT];
        struct pac  operands[OD];
    }
    moc[NC];
    struct   {
        int tval;
        char tname[ID];
    }
    ltok,ptok;

    for(i=0;i<NC;i++) {
        moc[i].numofoprnds = 0;
        moc[i].numofoprtrs = 0;
        moc[i].numofprdcts = 0;
        strcpy(moc[i].cname,"");
        }

    ptok.tval = 0;
    strcpy(ptok.tname,"");
    strcpy(moc[0].cname,"main");

    /* INITIALIZATION */

    noc = 1;
    cci = 0;

    while((tokval = yylex()) != 0)
    {
        ltok.tval = ptok.tval;
        strcpy(ltok.tname,ptok.tname);
        ptok.tval = tokval;
        strcpy(ptok.tname,yytext);

        switch(tokval)
        {
```

```
case MAIN :
     cci = 0;
     break;

case CLASS :
      if(ltok.tval == FRIEND)break;
     tokval = yylex();
     ltok.tval = ptok.tval;
     strcpy(ltok.tname,ptok.tname);
     ptok.tval = tokval;
     strcpy(ptok.tname,yytext);

     cci = noc;
     strcpy(moc[noc].cname,yytext);

     moc[cci].operators[ltok.tval-257].numtokoccur++;
     moc[cci].N1++;
     strcpy(moc[cci].operators[ltok.tval-257].tokenid,ltok.tname);
     noc++;
     break;

case DBCLN :
     index = 0;
     while(index <= moc[cci].numofoprnds &&
        strcmp(moc[cci].operands[index].tokenid,ltok.tname) != 0) index++;
     moc[cci].N2--;
     moc[cci].operands[index].numtokoccur-- ;
     cci = 0;
     while(strcmp(ltok.tname,moc[cci].cname)!=0 && cci <= noc+1)cci++;
      index = 0;
      while(index < moc[cci].numofoprnds &&
        strcmp(moc[cci].operands[index].tokenid,ltok.tname) != 0) index++;

     moc[cci].operands[index].numtokoccur++;
     moc[cci].N2++;
     strcpy(moc[cci].operands[index].tokenid,ltok.tname);
     break;
default :
     break;
}

if( (tokval == CASE   ) ||
   (tokval == FOR    ) ||
   (tokval == IF     ) ||
   (tokval == WHILE ) ||
   (tokval == AND_OP) ||
   (tokval == OR_OP ) )
     moc[cci].numofprdcts++;

if(tokval!=IDENTIFIER && tokval!=CONSTANT &&
    tokval != STRING_LITERAL && tokval != MAIN)
{
     moc[cci].operators[tokval-257].numtokoccur++;
     moc[cci].N1++;
     strcpy(moc[cci].operators[tokval-257].tokenid,yytext);
}
else
{
     index = 0;
     while(index < moc[cci].numofoprnds &&
           strcmp(moc[cci].operands[index].tokenid,yytext) != 0) index++;
     if(index < moc[cci].numofoprnds)
     {
           moc[cci].operands[index].numtokoccur++;
           moc[cci].N2++;
```

```
                    strcpy(moc[cci].operands[index].tokenid,yytext);
                }
                else
                {
                    moc[cci].operands[moc[cci].numofoprnds].numtokoccur++;
                    moc[cci].N2++;
                    strcpy(moc[cci].operands[moc[cci].numofoprnds].tokenid,yytext);
                    moc[cci].numofoprnds++;
                }
            }
        }

        cindex = 0;
        while(cindex < noc )
        {
            index = 0;

            while(index < OD){index++;
             if(moc[cindex].operators[index].numtokoccur > 0)moc[cindex].numofoprtrs++;}
            i = 0;
            printf("\n\n\n");
            printf("The CLASS name : %s\n\n",moc[cindex].cname);
            printf("The operators are as follows :-\nNo. of occurrences  The operator\n");
            while(i < OT)
            {
                if(moc[cindex].operators[i].numtokoccur != 0)
                    printf("       %d          %s\n",
                        moc[cindex].operators[i].numtokoccur,
                     moc[cindex].operators[i].tokenid);
                i++;
            }
            printf("\n\n\n");
            printf("The operands are as follows :-\nNo. of occurrences  The operand\n");
            i = 0;
            while(i < moc[cindex].numofoprnds && moc[cindex].operands[i].numtokoccur != 0)
            {
                printf("       %d          %s\n",moc[cindex].operands[i].numtokoccur,
                    moc[cindex].operands[i].tokenid);
                i++;
            }
            printf("\nN1 = %d, N2 = %d, eta1 = %d, eta2 = %d\n",moc[cindex].N1,moc[cindex].N2,
                  moc[cindex].numofoprtrs-1,moc[cindex].numofoprnds);
            printf("\n\n\n");
            N = (double) moc[cindex].N1 + moc[cindex].N2;
            eta = (double) moc[cindex].numofoprtrs + moc[cindex].numofoprnds - 2;
            V = N * log(eta)/log(2.0);
            printf("\n\n");
            printf("The Halstead's parameters are : \nN = %f\nV = %f\nETA = %f\n",N,V,eta);
            printf("\n\n\n");
            printf("The McCABE's parameter is : %d\n\n",moc[cindex].numofprdcts+1);
            cindex++;
        }
}
%}
D   [0-9]
I   [a-zA-Z]
Z   [0-9a-zA-Z]
H   [0-9a-fA-F]
O   [0-7]
P   [1-9]
L   [lL]
U   [uU]
E   [eE]
ESC [abfnrtv'"?\\]
X   [xX]
```

```
CE  \*/\/
A   "#"
C   "//"
F   "\n"
%%
"#"[ {F}]*"\n"                      { tracepos(); }
"//"[ {F}]*"\n"                     { tracepos(); }
"/*"[ {CE}]*"*/"                    { tracepos(); }
"asm"         {tracepos(); return(ASM);}
"auto"        {tracepos(); return(AUTO);}
"break"       {tracepos(); return(BREAK);}
"case"        {tracepos(); return(CASE);}
"char"        {tracepos(); return(CHAR);}
"class"       {tracepos(); return(CLASS);}
"continue"    {tracepos(); return(CONTINUE);}
"const"       {tracepos(); return(CONST);}
"default"     {tracepos(); return(DEFAULT);}
"delete"      {tracepos(); return(DELETE);}
"do"          {tracepos(); return(DO);}
"double"      {tracepos(); return(DOUBLE);}
"else"        {tracepos(); return(ELSE);}
"enum"        {tracepos(); return(ENUM);}
"extern"      {tracepos(); return(EXTERN);}
"float"       {tracepos(); return(FLOAT);}
"for"         {tracepos(); return(FOR);}
"friend"      {tracepos(); return(FRIEND);}
"goto"        {tracepos(); return(GOTO);}
"if"          {tracepos(); return(IF);}
"inline"      {tracepos(); return(INLINE);}
"int"         {tracepos(); return(INT);}
"long"        {tracepos(); return(LONG);}
"main"        {tracepos(); return(MAIN);}
"new"         {tracepos(); return(NEW);}
"operator"    {tracepos(); return(OPERATOR);}
"overload"    {tracepos(); return(OVERLOAD);}
"public"      {tracepos(); return(PUBLIC);}
"register"    {tracepos(); return(REGISTER);}
"return"      {tracepos(); return(RETURN);}
"short"       {tracepos(); return(SHORT);}
"signed"      {tracepos(); return(SIGNED);}
"sizeof"      {tracepos(); return(SIZEOF);}
"static"      {tracepos(); return(STATIC);}
"struct"      {tracepos(); return(STRUCT);}
"switch"      {tracepos(); return(SWITCH);}
"this"        {tracepos(); return(THIS);}
"typedef"     {tracepos(); return(TYPEDEF);}
"union"       {tracepos(); return(UNION);}
"unsigned"    {tracepos(); return(UNSIGNED);}
"virtual"     {tracepos(); return(VIRTUAL);}
"void"        {tracepos(); return(VOID);}
"volatile"    {tracepos(); return(VOLATILE);}
"while"       {tracepos(); return(WHILE);}
{I}({Z}|"_")*        {tracepos(); return(IDENTIFIER);}
0{X}{H}+{L}?{U}?          {tracepos(); return(CONSTANT);}
0{X}{H}+{U}?{L}?          {tracepos(); return(CONSTANT);}
0{O}+{L}?{U}?             {tracepos(); return(CONSTANT);}
0{O}+{U}?{L}?             {tracepos(); return(CONSTANT);}
{P}{D}*{L}?{U}?           {tracepos(); return(CONSTANT);}
{P}{D}*{U}?{L}?           {tracepos(); return(CONSTANT);}
\'([ '\n\\]|(\\({ESC}|{X}{H}{H}?{H}?|{O}{O}?{O}?)))+\'  {tracepos(); return(CONSTANT);}
\"([ "\n\\]|(\\({ESC}|{X}{H}{H}?{H}?|{O}{O}?{O}?)))+\"  {tracepos(); return(STRING_LITERAL);}
{D}+{E}{L}?               {tracepos(); return(CONSTANT);}
{D}*"."{D}+{E}?{L}?       {tracepos(); return(CONSTANT);}
{D}+"."{D}*{E}?{L}?       {tracepos(); return(CONSTANT);}
"..."         {tracepos(); return(ELLIPSIS);}
```

```
">>="        {tracepos(); return(RIGHT_ASSIGN);}
"<<="        {tracepos(); return(LEFT_ASSIGN);}
"+="         {tracepos(); return(ADD_ASSIGN);}
"-="         {tracepos(); return(SUB_ASSIGN);}
"*="         {tracepos(); return(MUL_ASSIGN);}
"/="         {tracepos(); return(DIV_ASSIGN);}
"%="         {tracepos(); return(MOD_ASSIGN);}
"&="         {tracepos(); return(AND_ASSIGN);}
"^="         {tracepos(); return(XOR_ASSIGN);}
"|="         {tracepos(); return(OR_ASSIGN);}
">>"         {tracepos(); return(RIGHT_OP);}
"<<"         {tracepos(); return(LEFT_OP);}
"++"         {tracepos(); return(INC_OP);}
"--"         {tracepos(); return(DEC_OP);}
"->"         {tracepos(); return(PTR_OP);}
"&&"         {tracepos(); return(AND_OP);}
"||"         {tracepos(); return(OR_OP);}
"<="         {tracepos(); return(LE_OP);}
">="         {tracepos(); return(GE_OP);}
"=="         {tracepos(); return(EQ_OP);}
"!="         {tracepos(); return(NE_OP);}
"::"         {tracepos(); return(DBCLN);}
";"          {tracepos(); return(SMCLN);}
"{"          {tracepos(); return(CRBRO);}
"}"          {tracepos(); /*return(CRBRC);*/}
","          {tracepos(); return(COMMA);}
":"          {tracepos(); return(FLCLN);}
"="          {tracepos(); return(EQUAL);}
"("          {tracepos(); return(PARNO);}
")"          {tracepos(); /*return(PARNC);*/}
"["          {tracepos(); return(SQBRO);}
"]"          {tracepos(); /*return(SQBRC);*/}
"."          {tracepos(); return(PRIOD);}
"&"          {tracepos(); return(AMPSD);}
"!"          {tracepos(); return(EXCLM);}
"~"          {tracepos(); return(NEGAT);}
"-"          {tracepos(); return(MINUS);}
"+"          {tracepos(); return(PLUSS);}
"*"          {tracepos(); return(MULTP);}
"/"          {tracepos(); return(DIVIS);}
"%"          {tracepos(); return(PRSNT);}
"<"          {tracepos(); return(LSTHN);}
">"          {tracepos(); return(GRTHN);}
"^"          {tracepos(); return(CARET);}
"|"          {tracepos(); return(ORSYM);}
"?"          {tracepos(); return(QUSTN);}
[ \t\v\n\f]  {tracepos(); }
%%
yywrap()
{
return(1);
}
int column = 0;
int tracepos()
{
  int i;
  for (i=0; yytext[i] != '\0'; i++)
    if (yytext[i] == '\n')
      column = 0;
    else if (yytext[i] == '\t')
      column += 8 - (column % 8);
    else
      column++;
  ECHO;
}
```

ABSTRACT

The object-oriented design strategy as both a problem decomposition and system development paradigm has made impressive inroads into the various areas of the computing sciences. Substantial development productivity improvements have been demonstrated in areas ranging from artificial intelligence to user interface design. However, there has been very little progress in the formal characterization of these productivity improvements and in the identification of the underlying cognitive mechanisms. The development and validation of models and metrics of this sort require large amounts of systematically-gathered structural and productivity data. There has, however, been a notable lack of systematically-gathered information on these development environments. A large part of this problem is attributable to the lack of a systematic programming environment evaluation methodology that is appropriate to the evaluation of object-oriented systems.

Consequently, the research presented in this document addresses the design, development, and evaluation of a systematic, extensible, and environment-independent methodology for the comparative evaluation of object-oriented programming environments. This methodology is intended to serve as a foundational element for supporting research into the impact of object-oriented software development environments and design strategies on the software development process and resultant software products. A systematic approach is defined for conducting the methodology with respect to the particular object-oriented programming environment under investigation. The evaluation of each environment is based on user performance of representative and well-specified

development tasks on well-characterized applications within the environment. Primary metrics needed to characterize the software applications under examination are also defined and monitored for subsequent use in the analysis and evaluation of the environments.

The major contributions of this work are as follows:

1. This research has formally established the primary metric data definitions that completely characterize the unique aspects of object-oriented software systems, including the inheritance lattice and messaging graph.

2. This research has established language-independent procedures for automatically capturing this primary metric data during an evaluation. These procedures have been shown to be instantiable in a representative set of object-oriented languages.

3. This research has established the fundamental characteristics of object-oriented software that indicate consistent applications of object-oriented design techniques, namely, that common capabilities are factored throughout the inheritance lattice and that individual objects focus on providing specific capabilities.

4. This research has defined a language-independent application domain-specific development paradigm based on these fundamental characteristics for highly interactive graphical applications.

5. This research has identified design principles for a programming environment evaluation methodology (PEEM) that ensure its applicability to object-oriented development environments. The PEEM design principles

unique to this work include the following: the requirement for primary metric data definitions that completely characterize the object-oriented characteristics of the software under evaluation, the requirement for the identification of relevant applications domain-specific development paradigms to support the validity and comparability of evaluative results, and the requirement for automatic capture of performance and primary metric data to ensure consistency and eliminate human bias.

6. Finally, this research has produced a systematic, extensible, and environment-independent programming environment evaluation methodology capable of supporting research into complexity models and metrics for object-oriented systems. The design principles, identified in contribution 5 above, establish the basis of the fundamental distinctions between exiting PEEMs and the PEEM developed as part of this research.

BIOGRAPHICAL SKETCH

Dennis R. Moreau was born in [redacted] on [redacted] [redacted] He received his B.S. and M.S. degrees in Computer Science from the University of Southwestern Louisiana, Lafayette, Louisiana in 1984 and 1986 respectively. He has worked as a professional systems analyst and systems consultant since 1979. Since 1984, he has served as the Senior NASA Research Assistant on the USL NASA Project of the Center for Advanced Computer Studies at the University of Southwestern Louisiana. During this time period, he has authored or co-authored 17 NASA research reports, conference publications, and journal articles. Additionally, he has served in capacities ranging from Project Manager to Technical Coordinator for over 115 hardware/software donation grants to the USL NASA Project.

5.20

1. Report No. IN-82	2. Government Accession No. 183590 ~~144393~~	3. Recipient's Catalog No.
4. Title and Subtitle 184P. USL/NGT-19-010-900: A PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY FOR OBJECT-ORIENTED SYSTEMS		5. Report Date DATE September 1987 OVERRIDE
		6. Performing Organization Code
7. Author(s) DENNIS R. MOREAU		8. Performing Organization Report No.
		10. Work Unit No.
9. Performing Organization Name and Address University of Southwestern Louisiana The Center for Advanced Computer Studies P.O. Box 44330 Lafayette, LA 70504-4330		11. Contract or Grant No. NGT-19-010-900
		13. Type of Report and Period Covered FINAL; 07/01/85 - 12/31/87
12. Sponsoring Agency Name and Address		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

This Working Paper Series entry represents the final Ph.D. Dissertation of Dennis R. Moreau, Senior USL NASA Research Assistant. The abstract of this research follows.

The object-oriented design strategy as both a problem decomposition and system development paradigm has made impressive inroads into the various areas of the computing sciences. Substantial development productivity improvements have been demonstrated in areas ranging from artificial intelligence to user interface design. However, there has been very little progress in the formal characterization of these productivity improvements and in the identification of the underlying cognitive mechanisms. The development and validation of models and metrics of this sort require large amounts of systematically-gathered structural and productivity data. There has, however, been a notable lack of systematically-gathered information on these development environments. A large part of this problem is attributable to the lack of a systematic programming environment evaluation methodology that is appropriate to the evaluation of object-oriented systems.

(Abstract continued on following page)

17. Key Words (Suggested by Author(s)) Programming Environment Evaluation Methodology, Object-Oriented Systems, PC-Based Research and Development	18. Distribution Statement

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 168	22. Price*

*For sale by the National Technical Information Service, Springfield, Virginia 22161